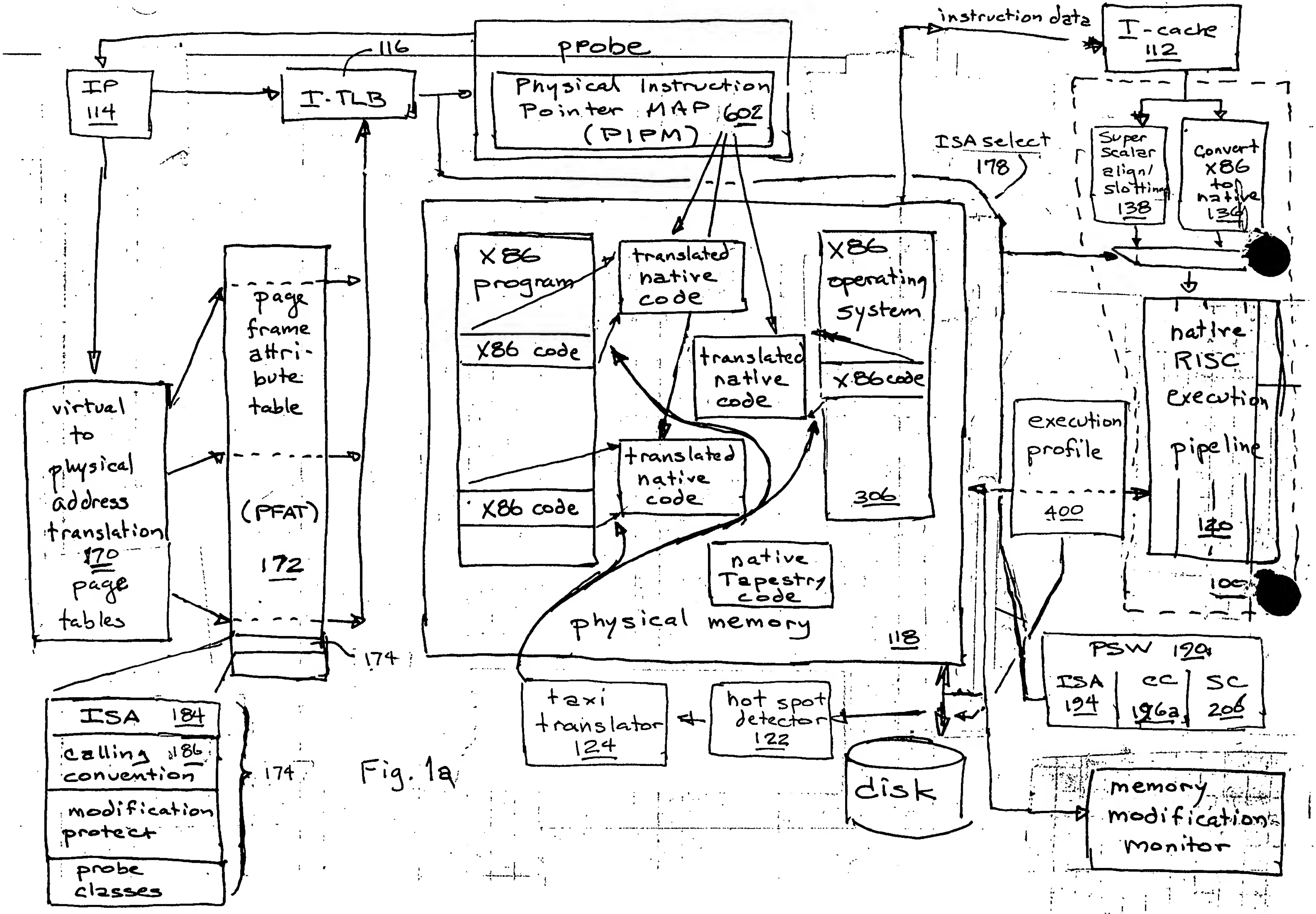

instruction data
I-cache 112
probe
Physical Instruction Pointer MAP 602 (PIPM)
116
IP 114
I-TLB
ISA select 178
Super Scalar align/ slotting 138
Convert X86 to native 136
native RISC execution pipeline 120
100
virtual to physical address translation 170 page tables
page frame attri-bute table (PFAT) 172
174
X86 program
X86 code
X86 code
translated native code
translated native code
translated native code
X86 operating system
X86 code
306
native Tapestry code
physical memory 118
execution profile 400
PSW 190
ISA 194
cc 196a
SC 206
taxi translator 124
hot spot detector 122
disk
memory modification monitor
ISA 184
calling convention 186
modification protect
probe classes
174

Fig. 1a

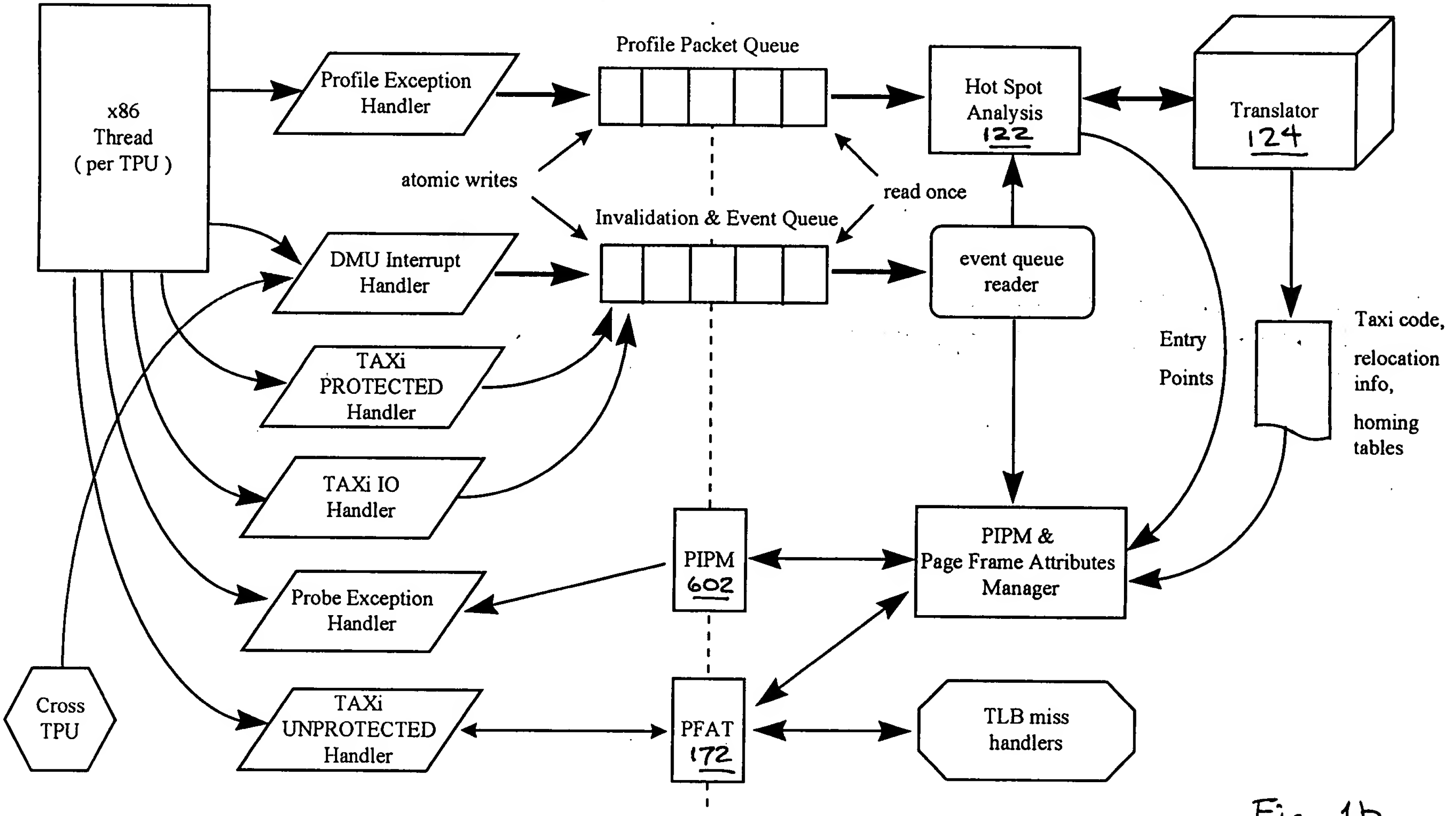
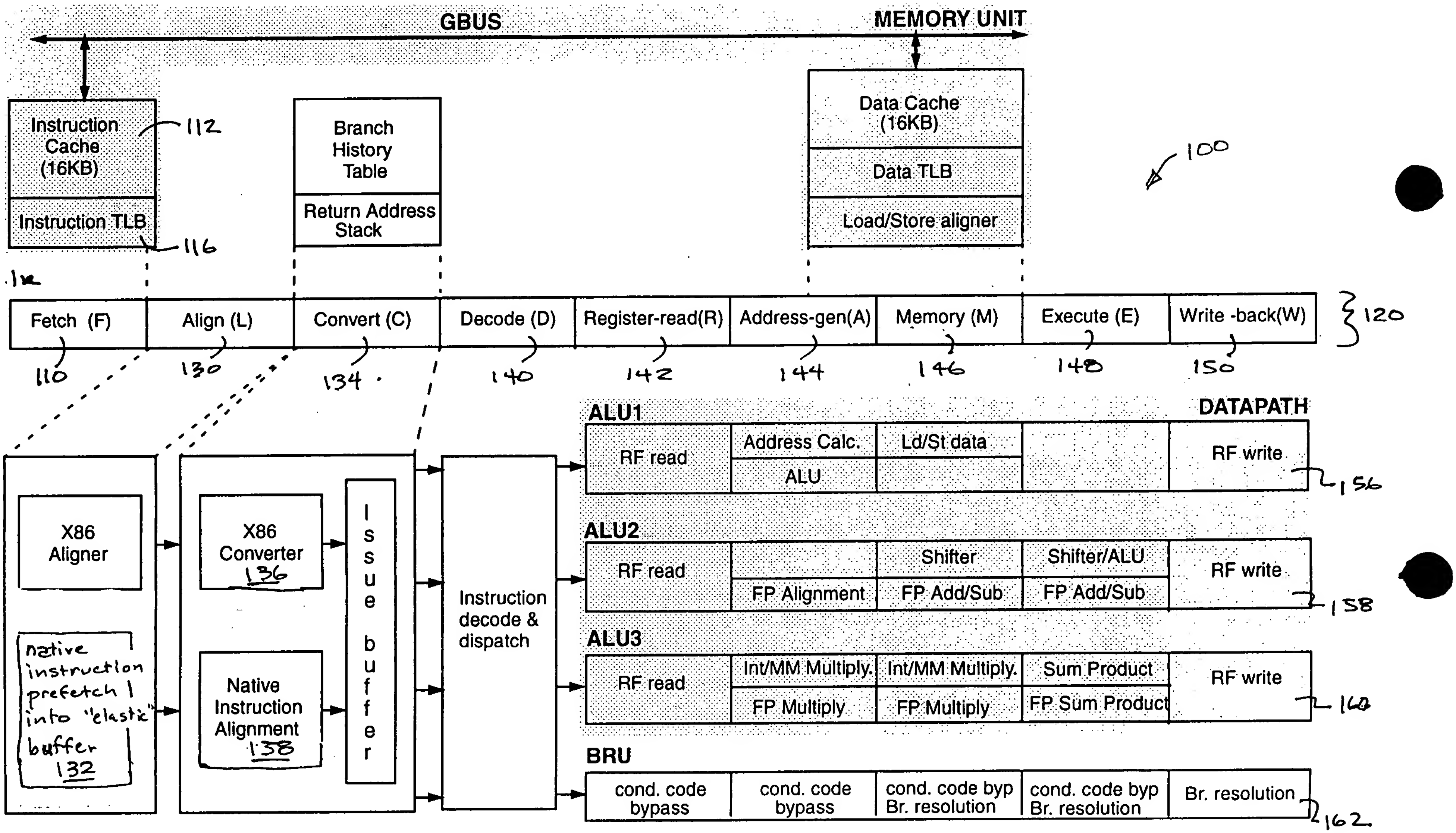

x86
Thread
(per TPU)
Profile Exception Handler
DMU Interrupt Handler
TAXi PROTECTED Handler
TAXi IO Handler
Probe Exception Handler
TAXi UNPROTECTED Handler
Cross TPU
atomic writes
Profile Packet Queue
Invalidation & Event Queue
read once
Hot Spot Analysis 122
Translator 124
event queue reader
Entry Points
Taxi code, relocation info, homing tables
PIPM & Page Frame Attributes Manager
PIPM 602
PFAT 172
TLB miss handlers

Fig. 1b

Fig. 1c


GBUS
MEMORY UNIT
Instruction Cache (16KB)
Instruction TLB
112
116
Branch History Table
Return Address Stack
Data Cache (16KB)
Data TLB
Load/Store aligner
100
Fetch (F)
Align (L)
Convert (C)
Decode (D)
Register-read(R)
Address-gen(A)
Memory (M)
Execute (E)
Write -back(W)
120
110
130
134
140
142
144
146
148
150
X86 Aligner
native instruction prefetch / into "elastic" buffer 132
X86 Converter 136
Native Instruction Alignment 138
Issue buffer
Instruction decode & dispatch
DATAPATH
ALU1
RF read
Address Calc.
ALU
Ld/St data
RF write
156
ALU2
RF read
FP Alignment
Shifter
FP Add/Sub
Shifter/ALU
FP Add/Sub
RF write
158
ALU3
RF read
Int/MM Multiply.
FP Multiply
Int/MM Multiply.
FP Multiply
Sum Product
FP Sum Product
RF write
160
BRU
cond. code bypass
cond. code bypass
cond. code byp Br. resolution
cond. code byp Br. resolution
Br. resolution
162

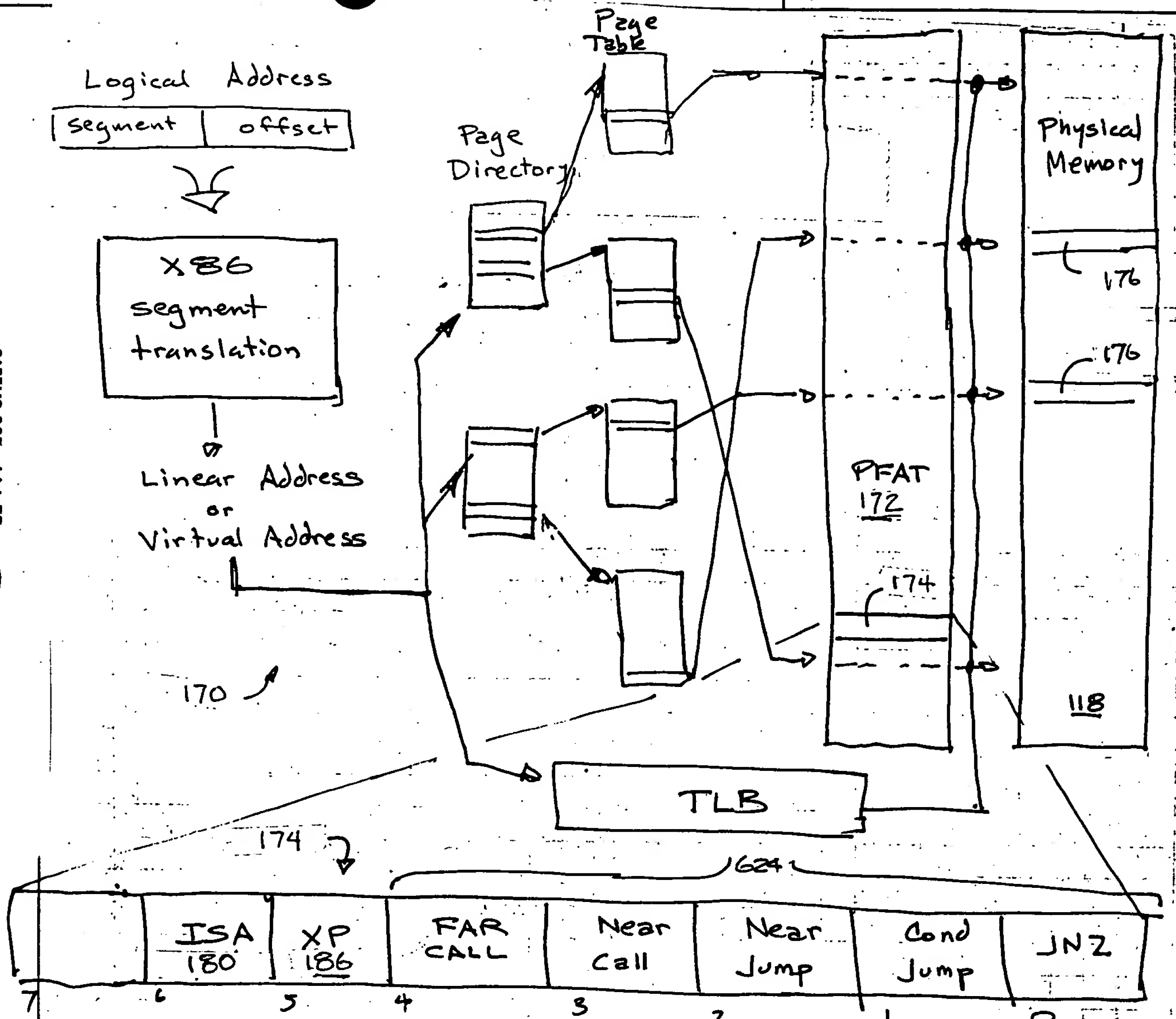


Fig. 1d - Memory Mapping

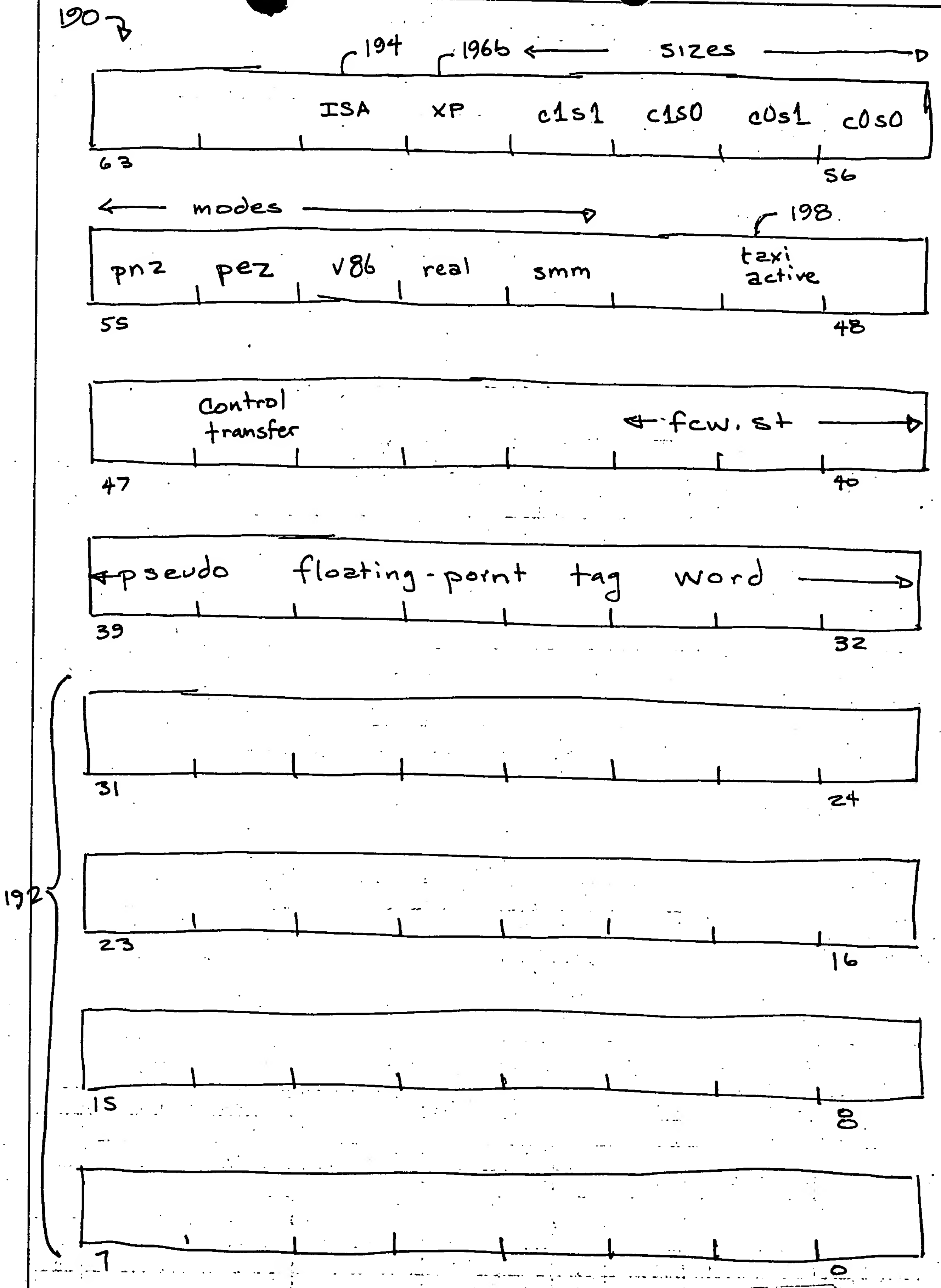

190
194
196b
sizes
ISA
XP
c1s1
c1s0
c0s1
c0s0
63
56
modes
198
pnz
pez
v86
real
smm
taxi active
55
48
control transfer
fcw.st
47
40
pseudo floating-point tag word
39
32
31
24
192
23
16
15
8
7
0

Fig. 1e

I-TLB property bits	Decoded property values			Interpretation	Instructions sent to:	Collect profile trace-packets?	Probe for translated code	I/O memory reference exceptions
	ISA 194	CC 200	Protected					
00	Tap	Tap	no	Native code observing native RISCy calling conventions	Native decoder	No	No	Fault if SEG.tio
01	Tap	x86	no	Native code observing x86 calling conventions	Native decoder	No	No	Fault if SEG.tio
10	x86	x86	no	x86 code, unprotected - *TAX!* profile collection only	x86 HW converter	If enabled	No	Trap if profiling
11	x86	x86	yes	x86 code, protected - *TAX!* code may be available	x86 HW converter	If enabled	Based on I-TLB probe attributes	Trap if profiling

180, 182
184, 186

184
186

Fig: 2a

Significance of the I-TLB property bits

204

	Transition (source => dest) ISA & CC property values	Handler Action
212	00 => 00	No transition exception
214	00 => 01	VECT_xxx_X86_CC exception - handler converts from native to x86 conventions
216	00 => 1x	VECT_xxx_X86_CC exception - handler converts from native to x86 conventions, sets up expected emulator and profiling state
218	01 => 00	VECT_xxx_TAP_CC exception - handler converts from x86 to native conventions
220	01 => 01	No transition exception
222	01 => 1x	VECT_X86_ISA exception [conditional based on PCW.X86_ISA_ENABLE flag] - sets up expected emulator and profiling state
224	1x => 00	VECT_xxx_TAP_CC exception - handler converts from x86 to native conventions
226	1x => 01	VECT_TAP_ISA exception [conditional based PCW.TAP_ISA_ENABLE flag] - no convention conversion necessary
228	1x => 10	No transition exception - [profile complete possible, probe possible]
230	1x => 11	No transition exception - [profile complete possible, probe NOT possible]

Fig. 2b

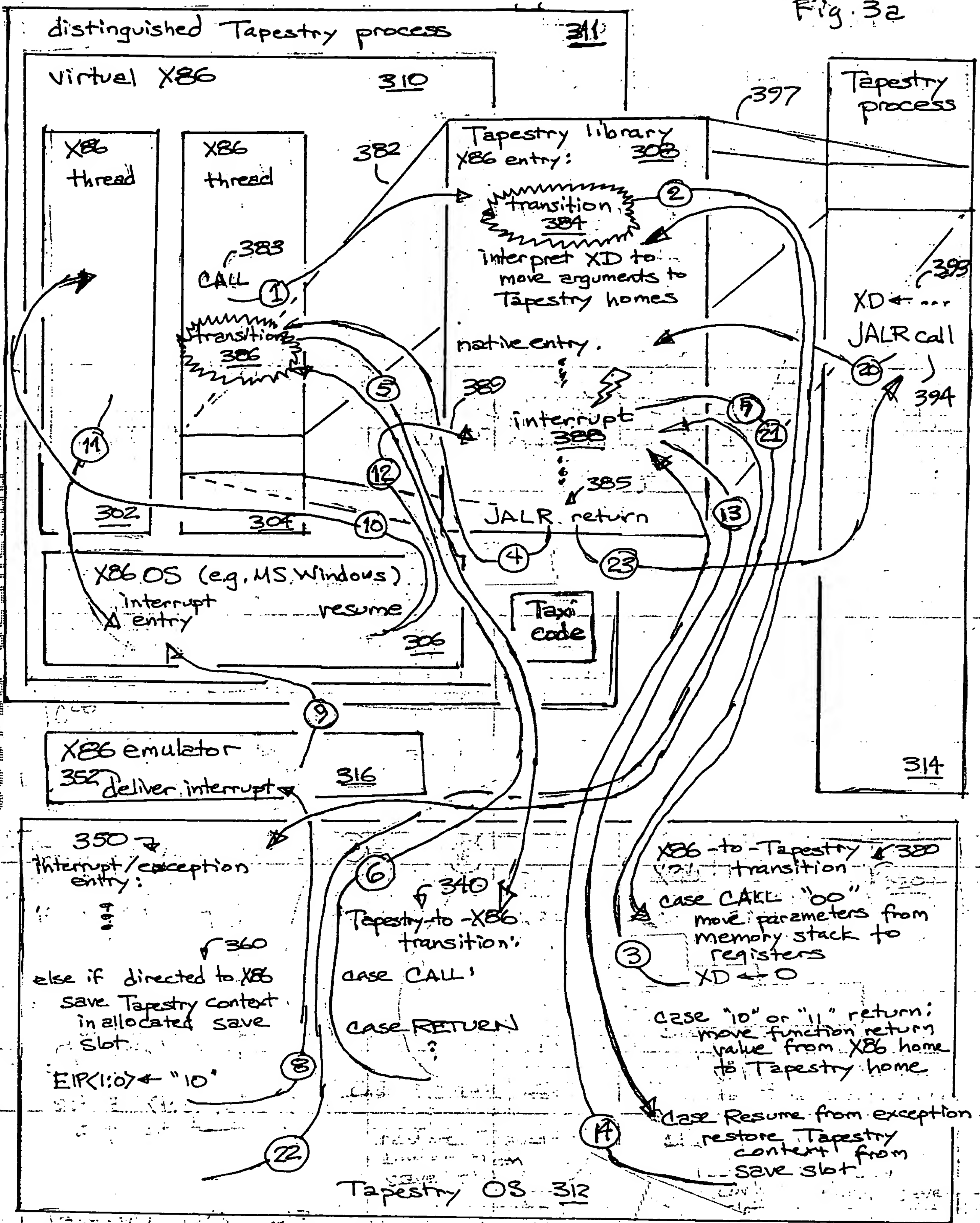
ISA & CC transition exception flow

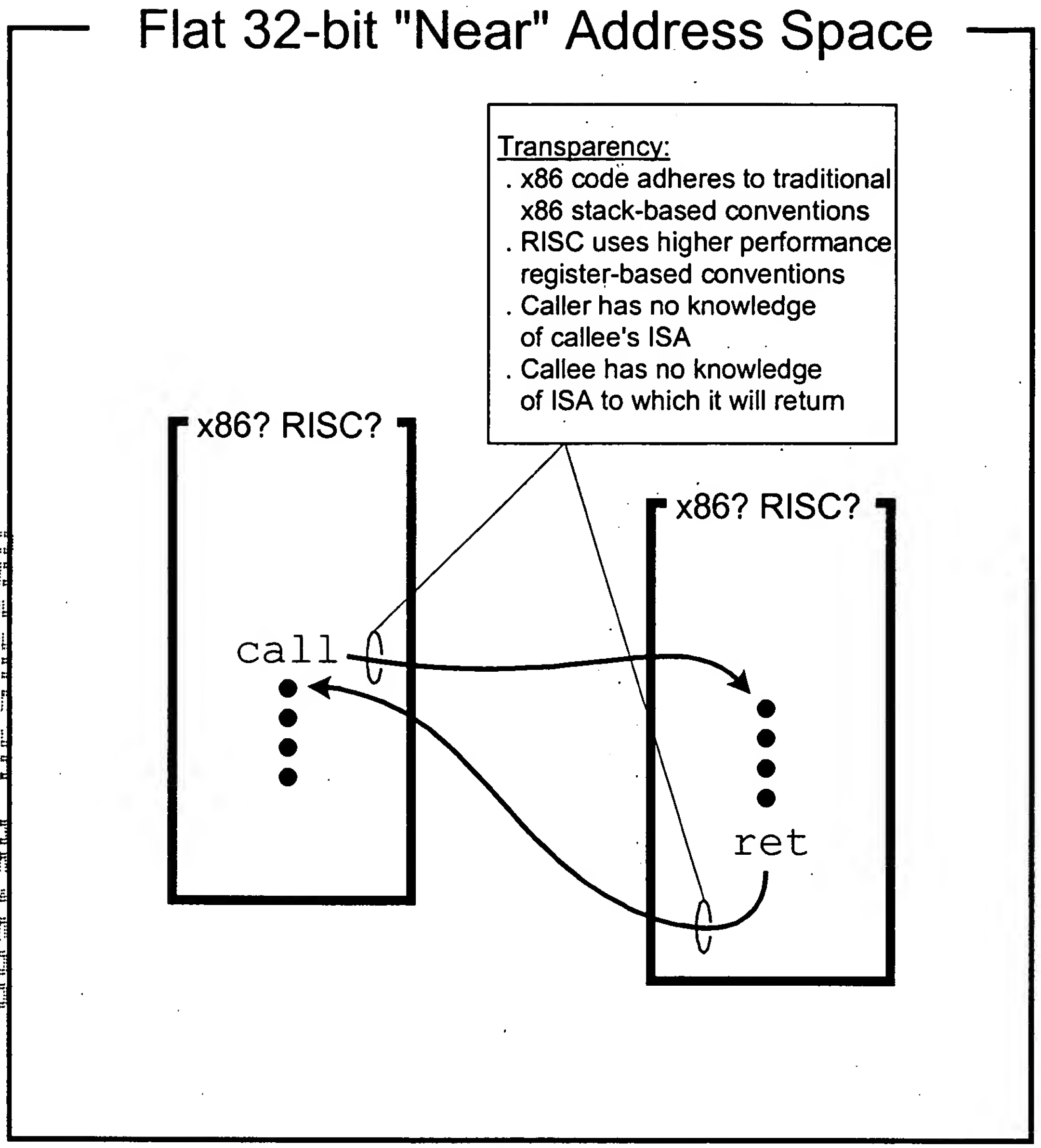
	name	description	type
242	VECT_call_X86_CC	push args, return address, set up x86 state	fault on target instruction
244	VECT_jump_X86_CC	set up x86 state	fault on target instruction
246	VECT_ret_no_fp_X86_CC	return value to eax:edx, set up x86 state	fault on target instruction
248	VECT_ret_fp_X86_CC	return value to x86 fp stack, set up x86 state	fault on target instruction
250	VECT_call_TAP_CC	x86 stack args, return address to registers	fault on target instruction
252	VECT_jump_TAP_CC	x86 stack args to registers	fault on target instruction
254	VECT_ret_no_fp_TAP_CC	return value to RV0	fault on target instruction
256	VECT_ret_any_TAP_CC	return type unknown, setup RV0 and RVDP	fault on target instruction

Fig. 2c

CC transition exceptions

Fig. 3a


distinguished Tapestry process
311
Virtuel X86
310
X86 thread
X86 thread
302
304
383
CALL
1
transition
386
5
11
12
10
4
382
Tapestry library
X86 entry:
308
transition
384
2
interpret XD to move arguments to Tapestry homes
native entry.
389
interrupt
388
385
JALR return
7
21
13
23
397
Tapestry process
395
XD ← ...
JALR call
20
394
314
X86 OS (e.g. MS Windows)
interrupt entry
resume
306
Taxi code
9
X86 emulator
352
deliver interrupt
316
350
Interrupt/exception entry:
360
else if directed to X86 save Tapestry context in allocated save slot
EIP(1:0) ← "10"
8
22
6
340
Tapestry-to-X86 transition:
case CALL:
case RETURN
?
X86-to-Tapestry transition
380
case CALL "00" move parameters from memory stack to registers
3
XD ← 0
case "10" or "11" return: move function return value from X86 home to Tapestry home
14
case Resume from exception restore Tapestry context from save slot
Tapestry OS 312



Transparency:

. x86 code adheres to traditional x86 stack-based conventions

. RISC uses higher performance register-based conventions

. Caller has no knowledge of callee's ISA

. Callee has no knowledge of ISA to which it will retum

Fig. 3b

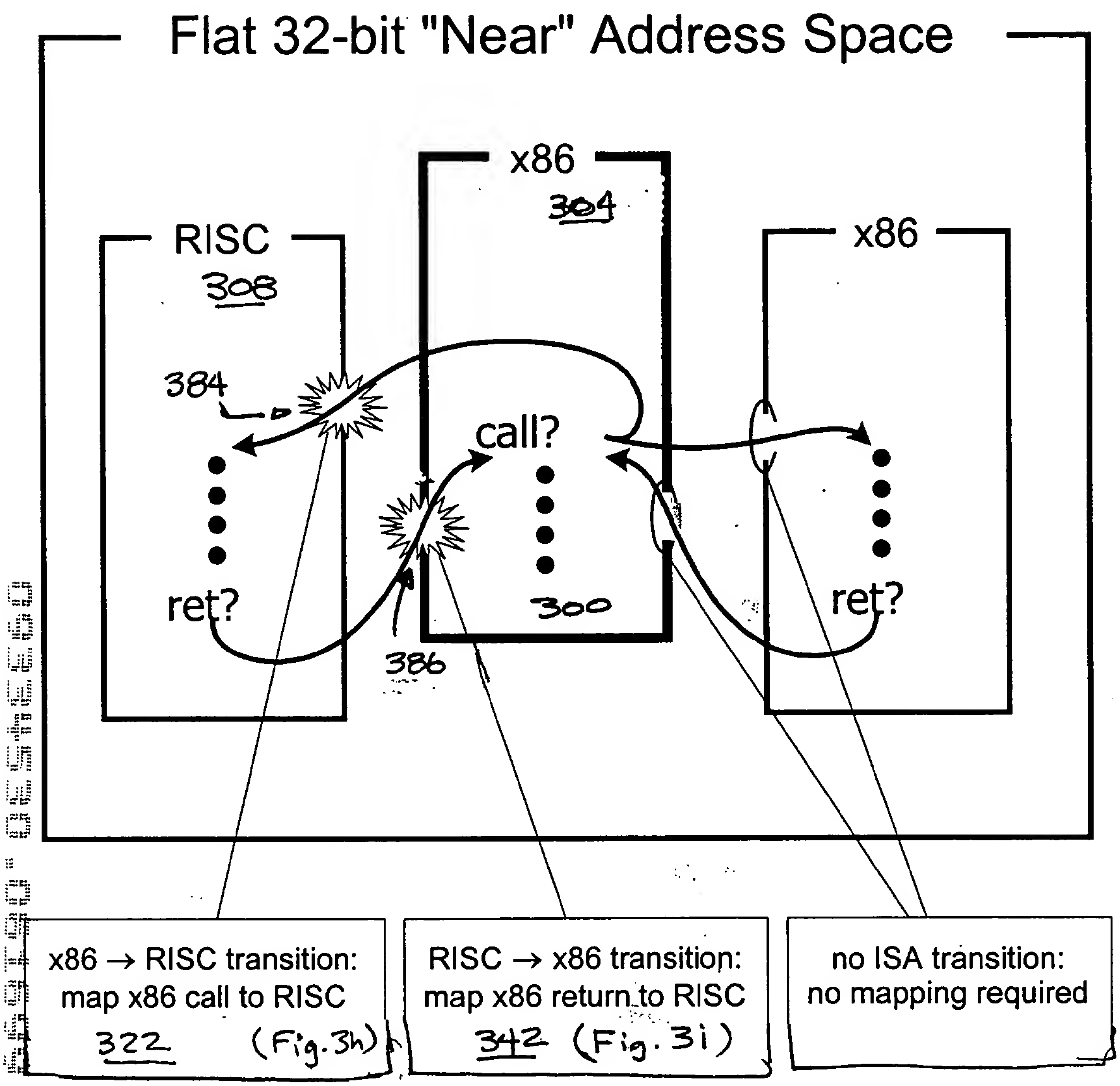

Flat 32-bit "Near" Address Space
x86
304
RISC
308
x86
384
call?
ret?
ret?
386
300
x86 → RISC transition:
map x86 call to RISC
322 (Fig. 3h)
RISC → x86 transition:
map x86 return to RISC
342 (Fig. 3i)
no ISA transition:
no mapping required

Fig. 3c

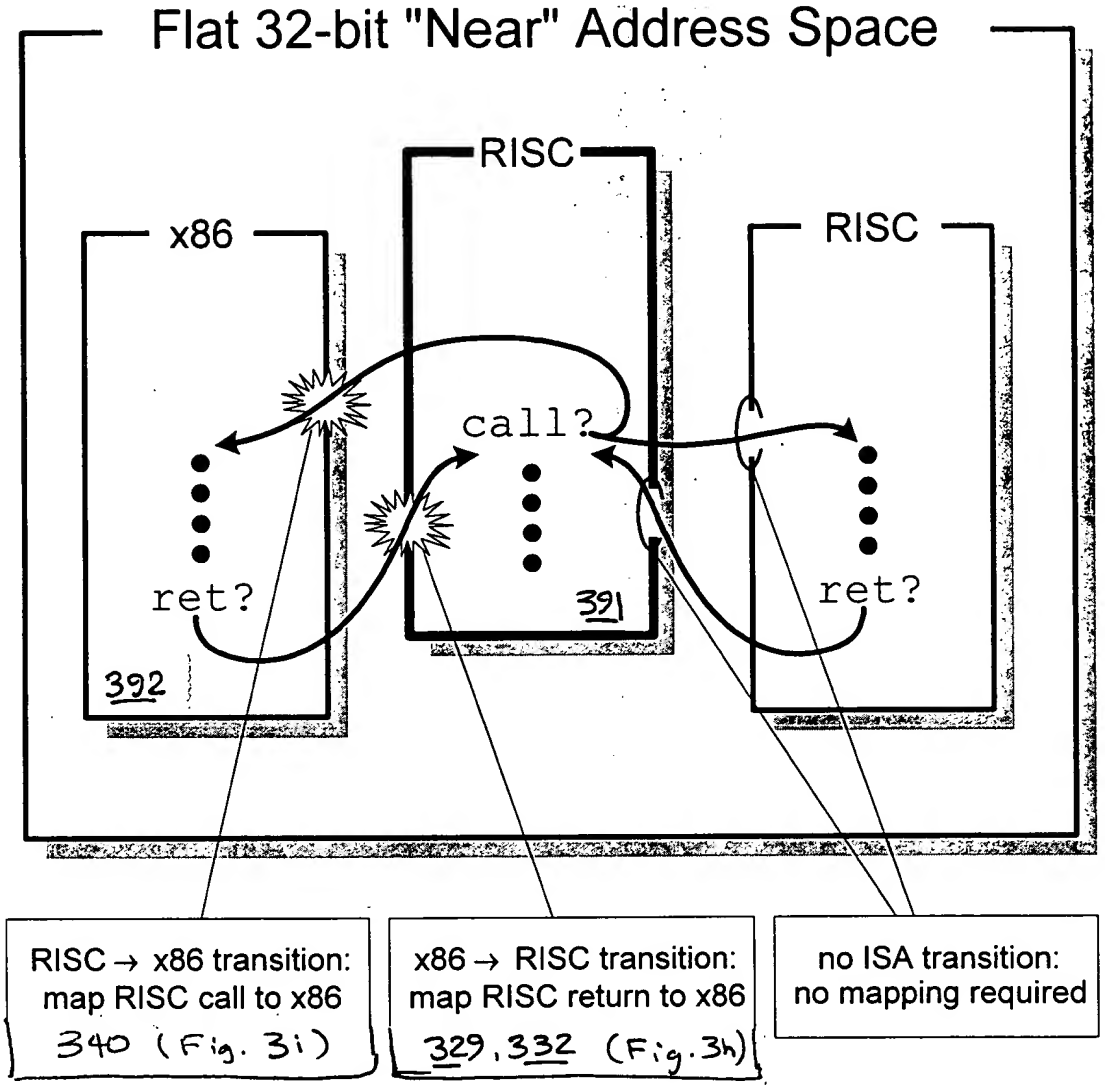

Flat 32-bit "Near" Address Space
RISC
x86
RISC
call?
ret?
ret?
391
392
RISC → x86 transition: map RISC call to x86
340 (Fig. 3i)
x86 → RISC transition: map RISC return to x86
329, 332 (Fig. 3h)
no ISA transition: no mapping required

Fig. 3d

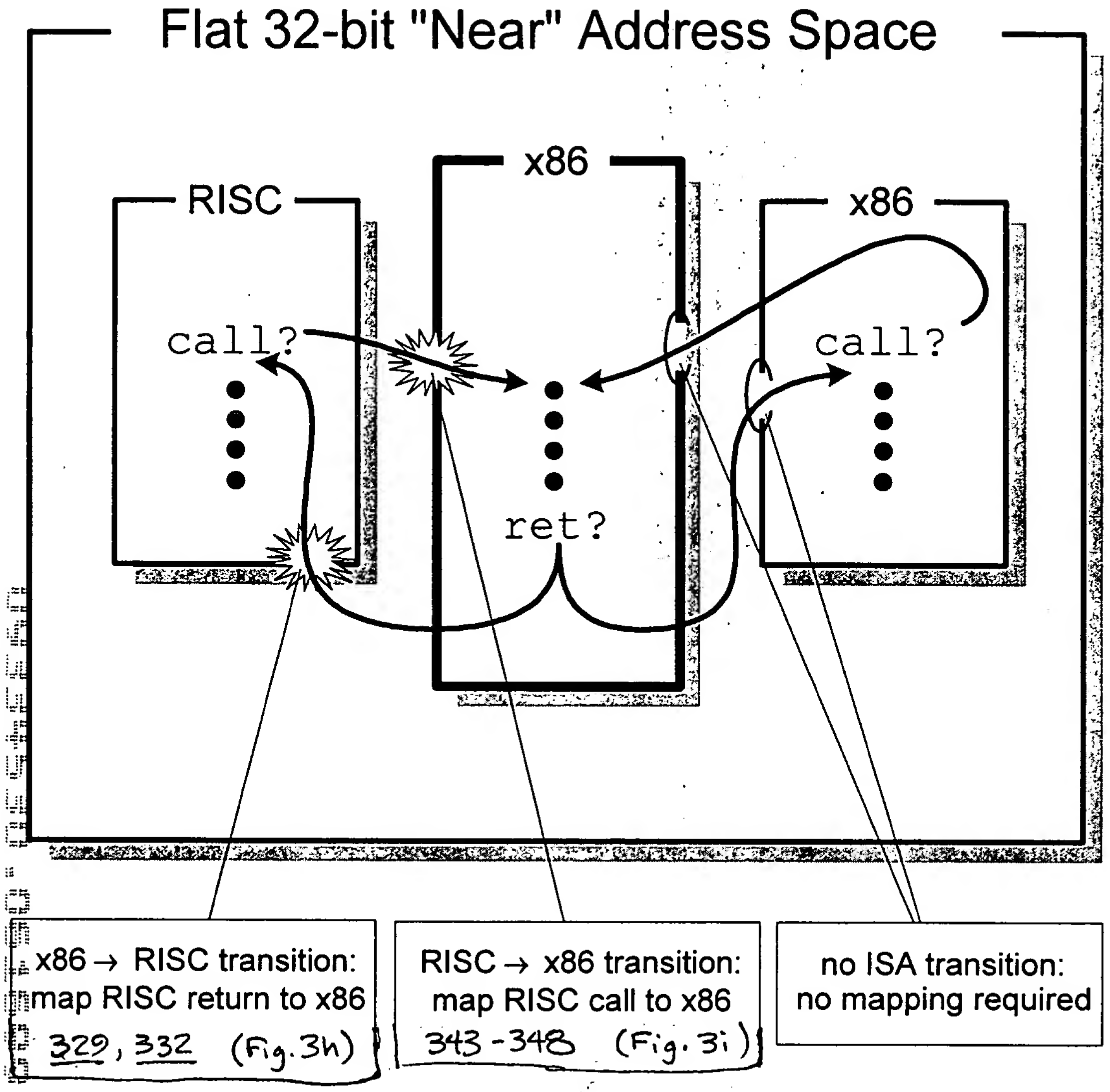
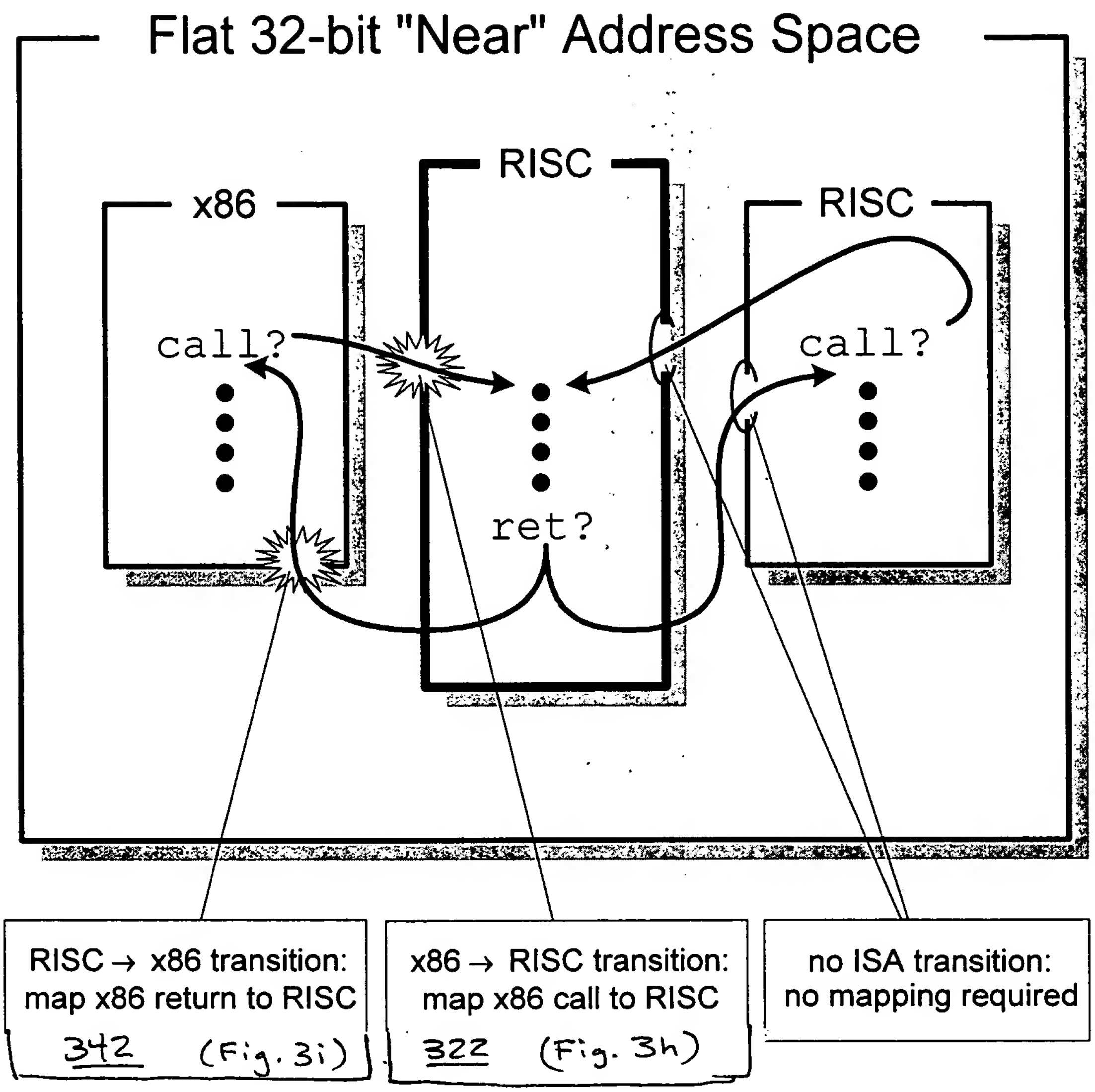

Flat 32-bit "Near" Address Space
RISC
x86
x86
call?
call?
ret?
x86 → RISC transition:
map RISC return to x86
329, 332 (Fig. 3h)
RISC → x86 transition:
map RISC call to x86
343-348 (Fig. 3i)
no ISA transition:
no mapping required

Fig. 3e



RISC → x86 transition: map x86 return to RISC 342 (Fig. 3i)

x86 → RISC transition: map x86 call to RISC 322 (Fig. 3h)

no ISA transition: no mapping required

Fig. 3f

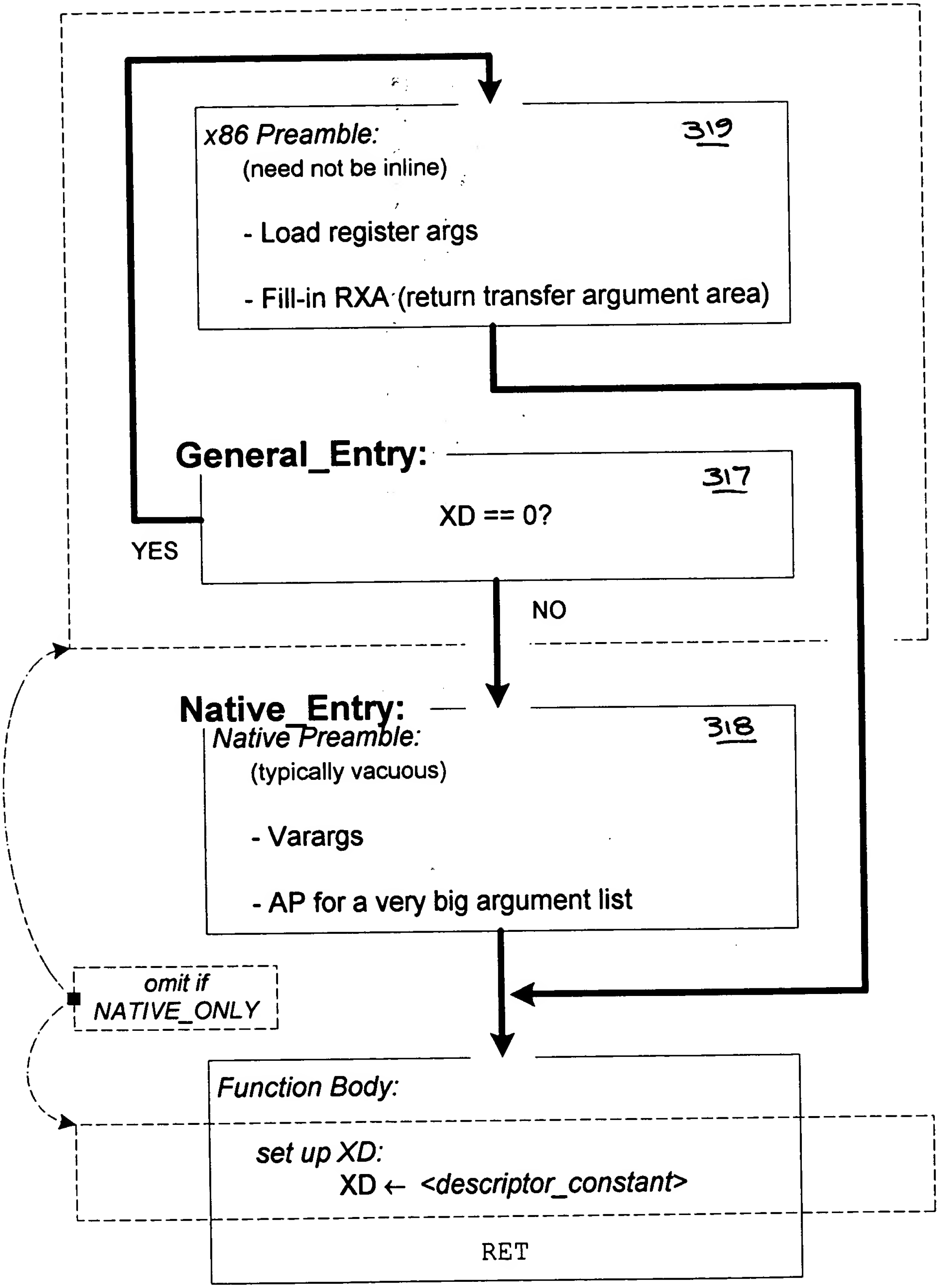

x86 Preamble:
(need not be inline)
- Load register args
- Fill-in RXA (return transfer argument area)
319
General_Entry:
317
XD == 0?
YES
NO
Native_Entry:
Native Preamble:
318
(typically vacuous)
- Varargs
- AP for a very big argument list
omit if
NATIVE_ONLY
Function Body:
set up XD:
XD ← <descriptor_constant>
RET

Fig. 3g

X86-to-Tapestry transition exception handler 320

```
// This handler is entered under the following conditions:
// 1. An x86 caller invokes a native function
// 2. An x86 function returns to a native caller
// 3. x86 software returns to or resumes an interrupted native function following
//    an external asynchronous interrupt, a processor exception, or a context switch
        321
dispatch on the two least-significant bits of the destination address     {
case "00"          // calling a native subprogram
    // copy linkage and stack frame information and call parameters from the memory
    // stack to the analogous Tapestry registers
    LR ← [SP++]        // set up linkage register  ~ 323                          } 322
    AP ← SP            // address of first argument  ~ 324
    SP ← SP - 8        // allocate return transfer argument area  ~ 326
    SP ← SP & (-32)    // round the stack pointer down to a 0 mode 32 boundary ~ 327
    XD ← 0             // inform callee that caller uses X86 calling conventions ~ 328
case "01"          // resuming an X86 thread suspended during execution of a native routine
    if  the redundant copies of the save slot number in EAX and EDX do not match or if    } 371
         the redundant copies of the timestamp in EBX:ECX and ESI:EDI do not match {
         // some form of bug or thread corruption has been detected
         goto TAPESTRY_CRASH_SYSTEM( thread-corruption-error-code )  ~ 372
    }
    save the EBX:ECX timestamp in a 64-bit exception handler temporary register  } 373     } 370
         (this will not be overwritten during restoration of the full native context)
    use save slot number in EAX to locate actual save slot storage  ~ 374
    restore full entire native context (includes new values for all x86 registers) ~ 375
    if save slot's timestamp does not match the saved timestamp {  ~ 376
         // save slot as been reallocated; save slot exhaustion has been detected
         goto TAPESTRY_CRASH_SYSTEM( save-slot-overwritten-error-code ) ~ 377
    }
    free the save slot ~ 378
case "10"          // returning from X86 callee to native caller, result already in registers
    RV0<63:32> ← edx<31:00>              // in case result is 64 bits  ~ 333      } 332
    convert the FP top-of-stack value from 80 bit X86 form to 64-bit form in RVDP ~ 334
    SP ← ESI                             // restore SP from time of call  ~ 337
case "11"          // returning from X86 callee to native caller, load large result from memory
    RV0..RV3 ← load 32 bytes from [ESI-32] // (guaranteed naturally aligned)  ~ 330   } 329
    SP ← ESI                             // restore SP from time of call  ~ 337
}
EPC ← EPC & -4         // reset the two low-order bits to zero ~ 336
RFE ~ 338
```

Fig. 3h

```
Tapestry-to-X86 transition exception handler                          340
    // This handler is entered under the following conditions:
    // 1. a native caller invokes an x86 function
    // 2. a native function returns to an x86 caller
    switch on XD<3:0> {  ~ 341

    XD_RET_FP:                      // result type is floating point
        F0/F1 ← FINFLATE.de( RVDP ) // X86 FP results are 80 bits
        SP ← from RXA save          // discard RXA, pad, args
        FPCW ← image after FINIT & push // FP stack has 1 entry
        goto EXIT

    XD_RET_WRITEBACK:               // store result to @RVA, leave RVA in eax
        RVA ← from RXA save         // address of result area
        copy decode(XD<8:4>) bytes from RV0..RV3 to [RVA]                 342
        eax ← RVA                   // X86 expects RVA in eax
        SP ← from RXA save          // discard RXA, pad, args
        FPCW ← image after FINIT        // FP stack is empty
        goto EXIT

    XD_RET_SCALAR:                  // result in eax:eda
        edx<31:00> ← eax<63:32>     // in case result is 64 bits
        SP ← from RXA save          // discard RXA, pad, args
        FPCW ← image after FINIT        // FP stack is empty
        goto EXIT

    XD_CALL_HIDDEN_TEMP:  // allocate 32 byte aligned hidden temp
        esi ← SP                    // stack cut back on return  ~ 343
        SP  ← SP – 32               // allocate max size temp    } 344
        RVA ← SP                    // RVA consumed later by RR  }
        LR<1:0> ← "11"              // flag address for return & reload  ~ 345
        goto CALL_COMMON

    default:                        // remaining XD_CALL_xxx encodings
        esi ← SP                    // stack cut back on return  ~ 343
        LR<1:0> ← "10"              // flag address for return   ~ 346
CALL_COMMON:
        interpret XD to push and/or reposition args   ~ 347
        [--SP] ← LR                 // push LR as return address  }
EXIT:                                                              } 348
        setup emulator context and profiling ring buffer pointer   }
    }
    RFE  ~ 349                      // to original target
}
```

Fig. 3i

```
interrupt/exception handler of Tapestry operating system:    350
     // Control vectors here when a synchronous exception or asynchronous interrupt is to be
     // exported to / manifested in an x86 machine.

// The interrupt is directed to something within the virtual X86, and thus there is a possibility
// that the X86 operating system will context switch. So we need to distinguish two cases:
//    either the running process has only X86 state that is relevant to save, or
//    there is extended state that must be saved and associated with the current machine context
//         (e.g., extended state in a Tapestry library call in behalf of a process managed by X86 OS)
if execution was interrupted in the converter – EPC.ISA == X86 {
          // no dependence on extended/native state possible hence no need to save any       } 351
          goto EM86_Deliver_Interrupt( interrupt-byte )
} else if EPC.Taxi_Active {
          // A Taxi translated version of some X86 code was running. Taxi will rollback to an
          // x86 instruction boundary. Then, if the rollback was induced by an asynchronous external   } 353
          // interrupt Taxi will deliver the appropriate x86 interrupt. Else, the rollback was induced
          // by a synchronous event so Taxi will resume execution in the converter, retriggering the
          // exception but this time will EPC.ISA == X86
          goto TAXi_Rollback( asynchronous-flag, interrupt-byte )
} else if EPC.EM86 {
          // The emulator has been interrupted. In theory the emulator is coded to allow for such    } 354
          // conditions and permits re-entry during long running routines (e.g. far call through a gate)
          // to deliver external interrupts
          goto EM86_Deliver_Interrupt( interrupt-byte )
} else {
          // This is the most difficult case – the machine was executing native Tapestry code on
          // behalf of an X86 thread. The X86 operating system may context switch. We must save
          // all native state and be able to locate it again when the x86 thread is resumed.
                    361
          allocate a free save slot; if unavailable free the save slot with oldest timestamp and try again   } 360
          save the entire native state (both the X86 and the extended state)     } 362
          save the X86 EIP in the save slot
          overwrite the two low-order bits of EPC with "01" (will become X86 interrupt EIP)   363
          store the 64-bit timestamp in the save slot, in the X86 EBX:ECX register pair (and,    } 364
                    for further security, store a redundant copy in the X86 ESI:EDI register pair)
          store the a number of the allocated save slot in the X86 EAX register (and, again for    } 365
                    further security, store a redundant copy in the X86 EDX register)
          goto EM86_Deliver_Interrupt( interrupt-byte )   369
}
```

350

Fig. 3j

```
typedef struct {
    save_slot_t *     newer;           // pointer to next-most-recently-allocated save slot  } 379c
    save_slot_t *     older;           // pointer to next-older save slot                    }
    unsigned int64    epc;             // saved exception PC/IP                              }       }
    unsigned int64    pcw;             // saved exception PCW (program control word)         } 356   } 355
    unsigned int64    registers[63];   // save the 63 writeable general registers            }       }
    ...                                // other words of Tapestry context                    }       }
    timestamp_t       timestamp;       // timestamp to detect buffer overrun  ~ 358                  }
    int               save_slot_ID;    // ID number of the save slot  ~ 357                          }
    boolean           save_slot_is_full;     // full / empty flag  ~ 359                             }
} save_slot_t;


save_slot_t *         save_slot_head;        // pointer to the head of the queue  ~ 379a
save_slot_t *         save_slot_tail;        // pointer to the tail of the queue  ~ 379b
```

system initialization

reserve several pages of unpaged memory for save slots

Fig. 3k

Fig. 3L

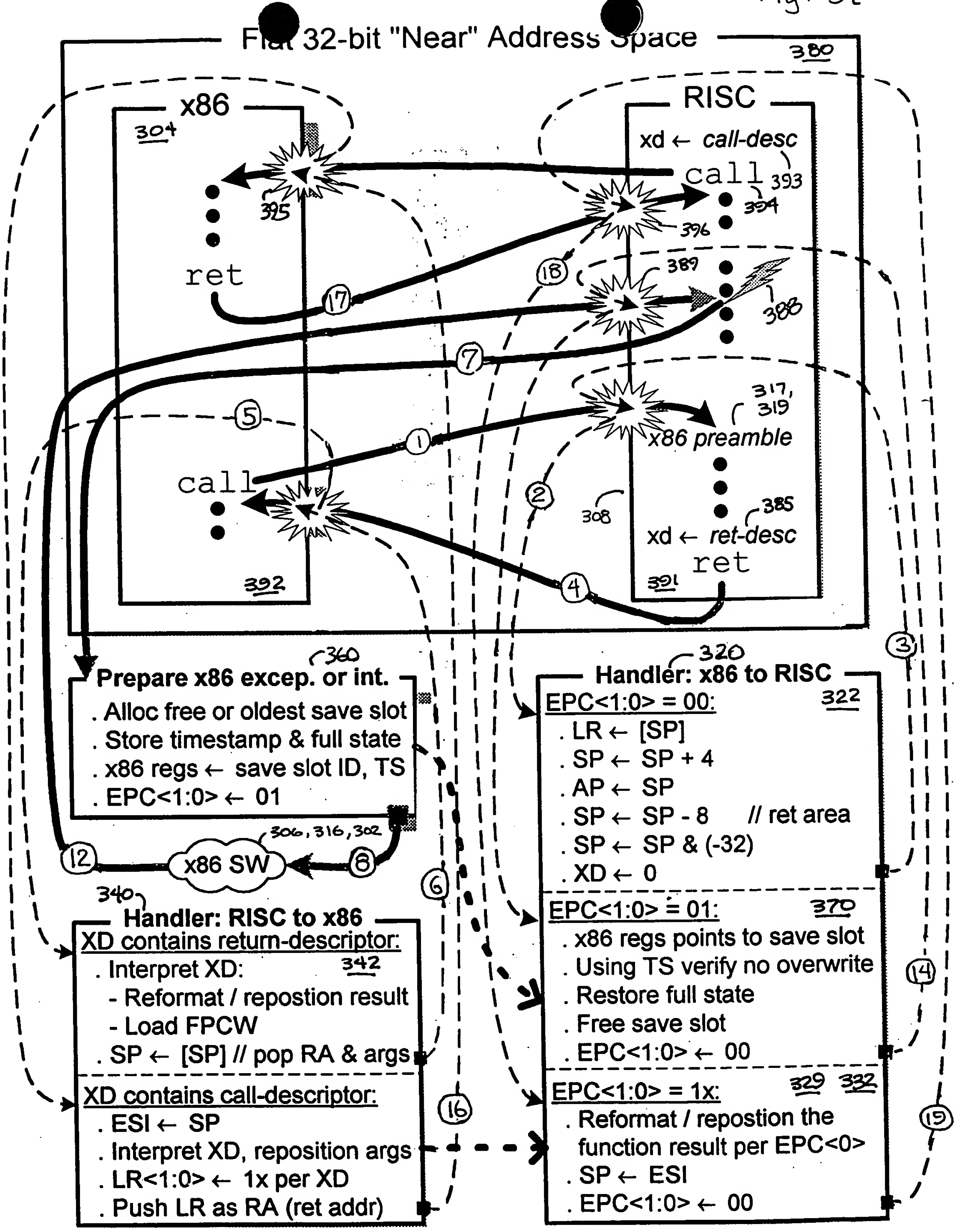

Flat 32-bit "Near" Address Space
380
x86
304
RISC
xd ← call-desc
call
393
394
395
396
ret
17
18
389
388
7
5
317, 319
x86 preamble
1
call
2
308
385
xd ← ret-desc
ret
4
392
391
3
360
Prepare x86 excep. or int.
. Alloc free or oldest save slot
. Store timestamp & full state
. x86 regs ← save slot ID, TS
. EPC<1:0> ← 01
306, 316, 302
12
x86 SW
8
6
340
Handler: RISC to x86
XD contains return-descriptor:
. Interpret XD:
342
- Reformat / repostion result
- Load FPCW
. SP ← [SP] // pop RA & args
XD contains call-descriptor:
. ESI ← SP
. Interpret XD, reposition args
. LR<1:0> ← 1x per XD
. Push LR as RA (ret addr)
16
320
Handler: x86 to RISC
EPC<1:0> = 00:
322
. LR ← [SP]
. SP ← SP + 4
. AP ← SP
. SP ← SP - 8 // ret area
. SP ← SP & (-32)
. XD ← 0
EPC<1:0> = 01:
370
. x86 regs points to save slot
. Using TS verify no overwrite
. Restore full state
. Free save slot
. EPC<1:0> ← 00
14
EPC<1:0> = 1x:
329 332
. Reformat / repostion the
function result per EPC<0>
. SP ← ESI
. EPC<1:0> ← 00
19

Fig. 3L

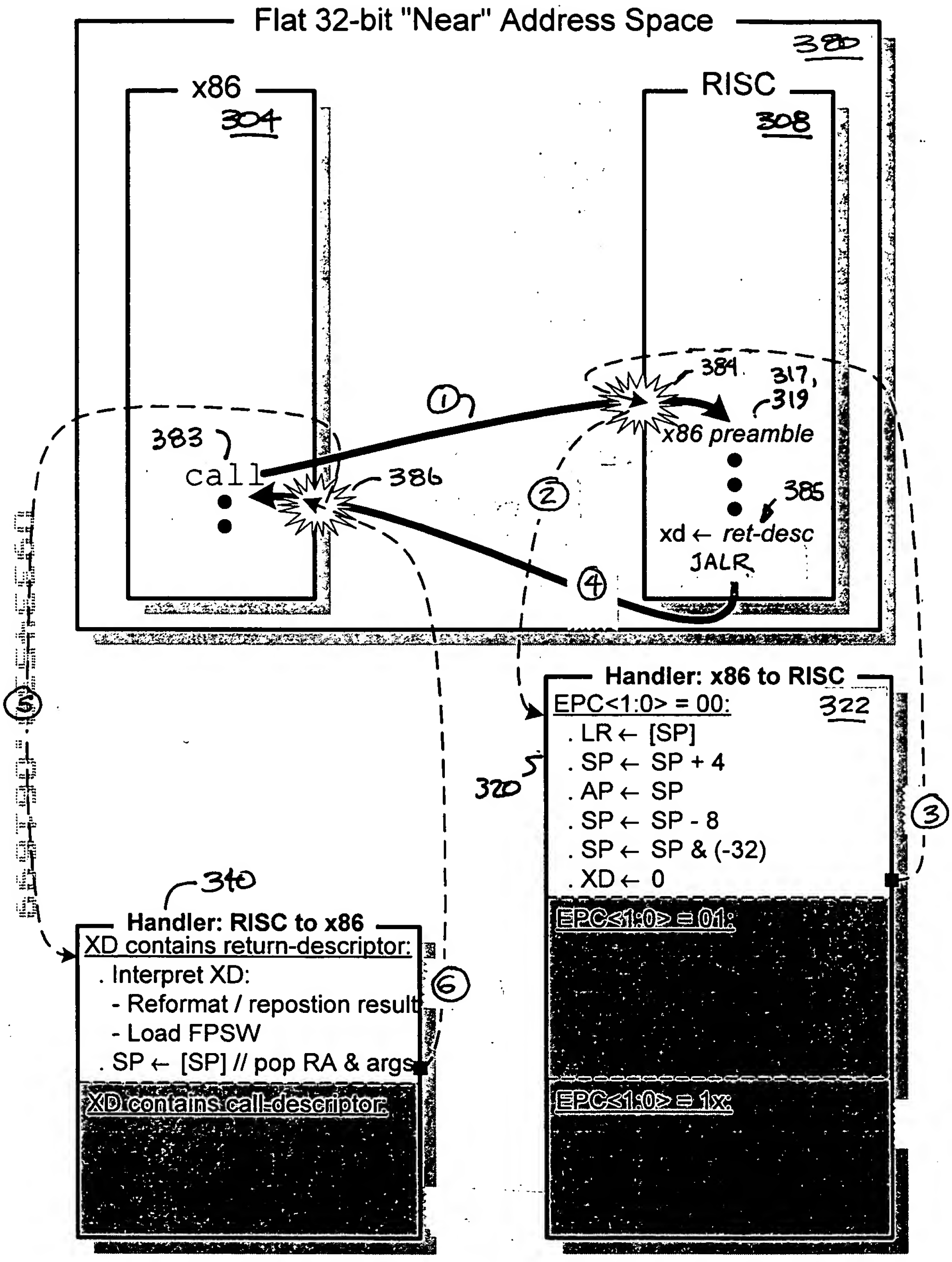
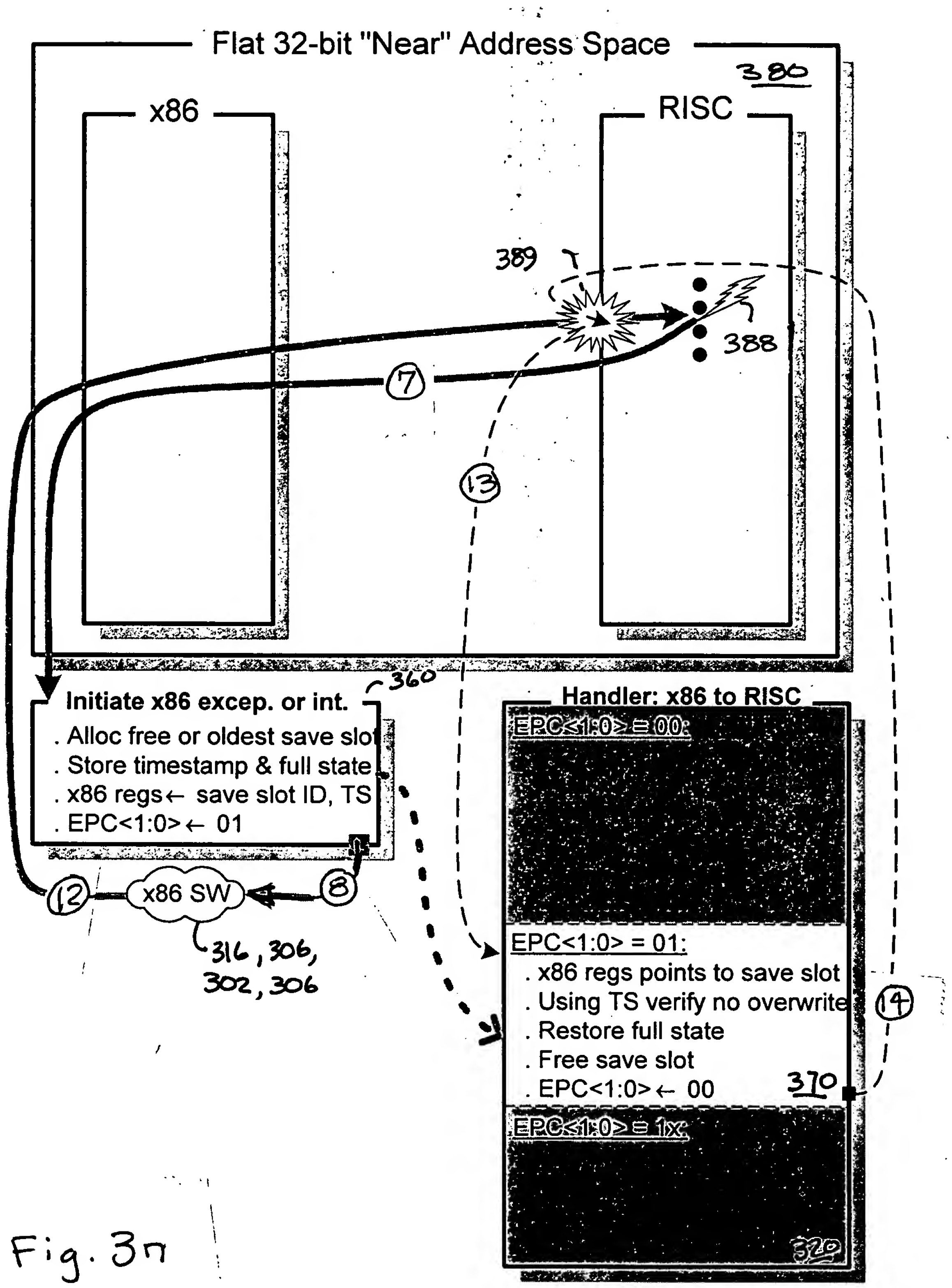

Flat 32-bit "Near" Address Space
380
x86
304
RISC
308
1
384
317,
319
x86 preamble
383
call
386
2
385
xd ← ret-desc
JALR
4
5
3
Handler: x86 to RISC
322
EPC<1:0> = 00:
. LR ← [SP]
. SP ← SP + 4
. AP ← SP
. SP ← SP - 8
. SP ← SP & (-32)
. XD ← 0
320
EPC<1:0> = 01:
EPC<1:0> = 1x:
340
Handler: RISC to x86
XD contains return-descriptor:
. Interpret XD:
- Reformat / repostion result
- Load FPSW
. SP ← [SP] // pop RA & args
6
XD contains call-descriptor:

Fig. 3m



Flat 32-bit "Near" Address Space
380
x86
RISC
389
388
7
13
14
Initiate x86 excep. or int.
360
. Alloc free or oldest save slot
. Store timestamp & full state
. x86 regs ← save slot ID, TS
. EPC<1:0> ← 01
12
x86 SW
8
316, 306, 302, 306
Handler: x86 to RISC
EPC<1:0> = 00:
EPC<1:0> = 01:
. x86 regs points to save slot
. Using TS verify no overwrite
. Restore full state
. Free save slot
. EPC<1:0> ← 00
370
EPC<1:0> = 1x:
320

Fig. 3n

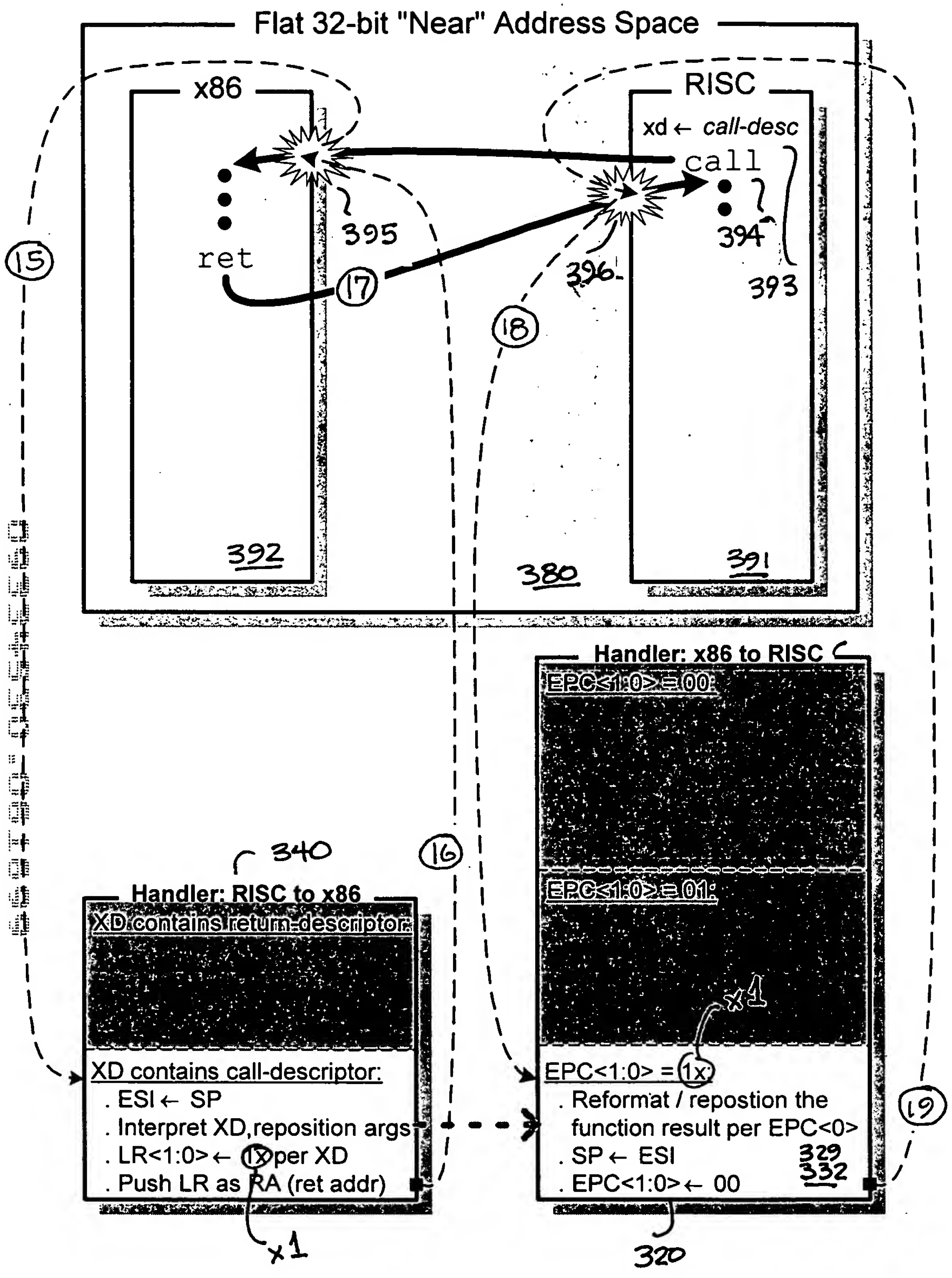
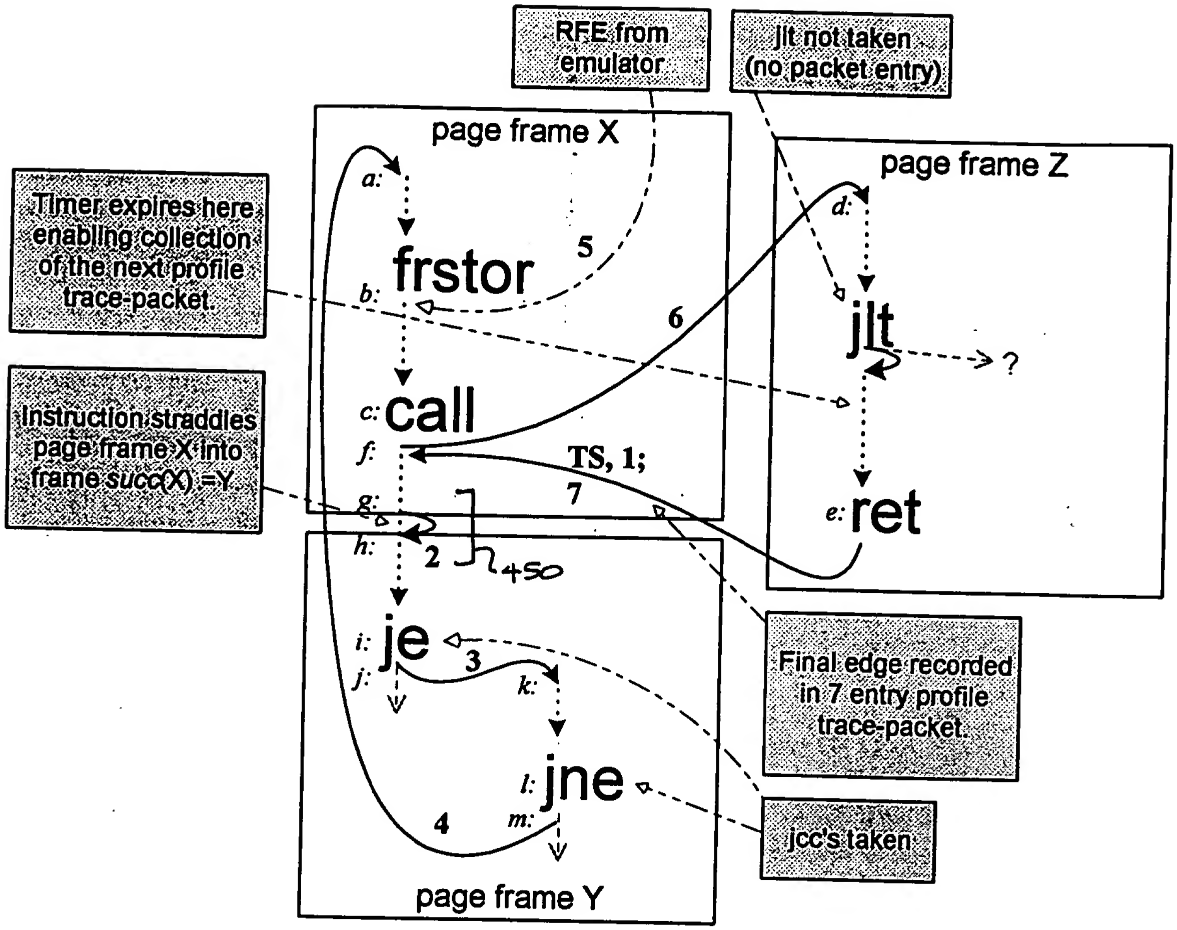

Flat 32-bit "Near" Address Space
x86
RISC
xd ← call-desc
call
ret
395
394
393
396
392
391
380
15
16
17
18
19
Handler: x86 to RISC
EPC<1:0> = 00:
EPC<1:0> = 01:
x1
EPC<1:0> = 1x:
. Reformat / repostion the function result per EPC<0>
. SP ← ESI
. EPC<1:0> ← 00
329
332
320
340
Handler: RISC to x86
XD contains return-descriptor:
XD contains call-descriptor:
. ESI ← SP
. Interpret XD,reposition args
. LR<1:0> ← 1x per XD
. Push LR as RA (ret addr)
x1

Fig. 30


RFE from emulator
jlt not taken (no packet entry)
page frame X
page frame Z
Timer expires here enabling collection of the next profile trace-packet.
a:
frstor
5
b:
6
d:
jlt
?
c: call
f:
TS, 1;
7
e: ret
Instruction straddles page frame X into frame succ(X) =Y.
g:
h:
2
450
i: je
3
j:
k:
Final edge recorded in 7 entry profile trace-packet.
l: jne
m:
4
jcc's taken
page frame Y

7 entry trace packet

420

Entry	Event Code	Done Addr	Next Addr	
	64 bit time stamp			
1	ret	x86 context	*phys X:f*	~ 430
2	new page	*phys Y:g*	*phys Y:h*	~ 440, 454
3	jcc forward	*phys Y:i*	*phys Y:k*	~ 440
4	jnz backward	*phys Y:l*	*phys X:a*	~ 440
5	seq; env change	x86 context	*phys X:b*	~ 430
6	ip-rel near call	*phys X:c*	*phys Z:d*	~ 440
7	near ret	*phys Z:e*	*phys X:f*	~ 440

Fig. 4a

412

410

404

Source	Code 402	Event	Reuse event code 414	Profileable event 416	Initiate packet 418	Probeable event 610	Probe event bit - ITLB probe attribute or Emulator probe 612
RFE (Context_at_Point entry)	0.0000	Default (x86 transparent) event, reuse all converter values	yes		no		
	0.0001	Simple x86 instruction completion (reuse event code)	yes		no		
	0.0010	Probe exception failed	yes		no		reuse event code
	0.0011	Probe exception failed, reload probe timer	yes		no		
	0.0100	flush event	no	no	no	no	-
	0.0101	Sequential; execution environment changed – force event	no	yes	no	no	-
	0.0110	Far RET	no	yes	yes	no	-
	0.0111	IRET	no	yes	no	no	-
	0.1000	Far CALL	no	yes	yes	yes	Far call
	0.1001	Far JMP	no	yes	yes	no	-
	0.1010	Special; emulator execution, supply extra instruction data[a]	no	yes	no	no	-
	0.1011	Abort profile collection	no	no	no	no	-
	0.1100	x86 synchronous/asynchronous interrupt w/probe (GRP 0)	no	yes	yes	yes	Emulator probe
	0.1101	x86 synchronous/asynchronous interrupt (GRP 0)	no	yes	yes	no	-
	0.1110	x86 synchronous/asynchronous interrupt w/probe (GRP 1)	no	yes	yes	yes	Emulator probe
	0.1111	x86 synchronous/asynchronous interrupt (GRP 1)	no	yes	yes	no	-
Converter (Near_Edge entry)	1.0000	IP-relative JNZ forward (opcode: 75, 0F 85)	no	yes	yes	no	-
	1.0001	IP-relative JNZ backward (opcode: 75, 0F 85)	no	yes	yes	yes	Jnz
	1.0010	IP-relative conditional jump forward - (Jcc, Jcxz, loop)	no	yes	yes	no	-
	1.0011	IP-relative conditional jump backward - (Jcc, Jcxz, loop)	no	yes	yes	yes	Cond jump
	1.0100	IP-relative, near JMP forward (opcode: E9, EB)	no	yes	yes	no	-
	1.0101	IP-relative, near JMP backward (opcode: E9, EB)	no	yes	yes	yes	Near jump
	1.0110	RET/ RET imm16 (opcode C3, C2 /w)	no	yes	yes	no	-
	1.0111	IP-relative, near CALL (opcode: E8)	no	yes	yes	yes	Near call
	1.1000	REPE/REPNE CMPS/SCAS (opcode: A6, A7, AE, AF)	no	yes	no	no	-
	1.1001	REP MOVS/STOS/LDOS (opcode: A4, A5, AA, AB, AC, AD)	no	yes	no	no	-
	1.1010	Indirect near JMP (opcode: FF /4)	no	yes	yes	no	-
	1.1011	Indirect near CALL (opcode: FF /2)	no	yes	yes	yes	Near call
	1.1100	load from I/O memory (TLB.asi != 0) { *not used in T1* }	no	yes	no	no	-
	1.1101	available for expansion	no	no	no	no	
	1.1110	Default converter event; sequential 406	no	no	no	no	-
	1.1111	New page (instruction ends on last byte of a page frame or straddles across a page frame boundary) 408	no	yes	no	no	-

a. Used by emulator for new x86 opcodes. Extra information supplied in *Taxi_Control.special_opcode* bits.

Fig. 4b

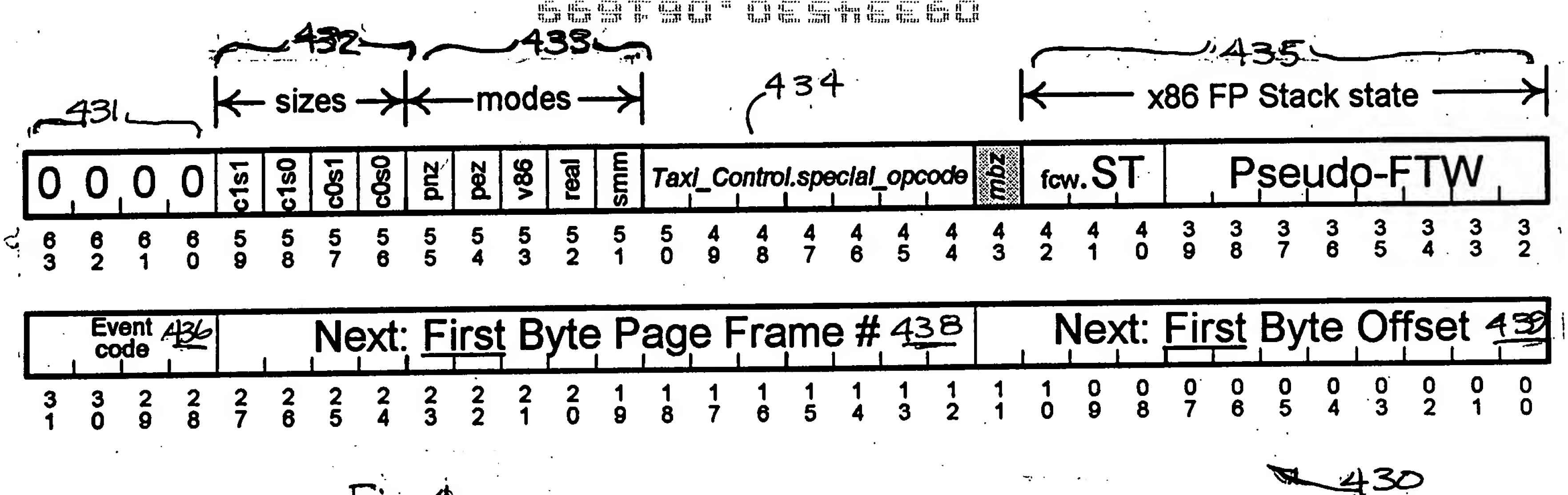


Fig. 4c *Context_At_Point profile trace-packet entry*

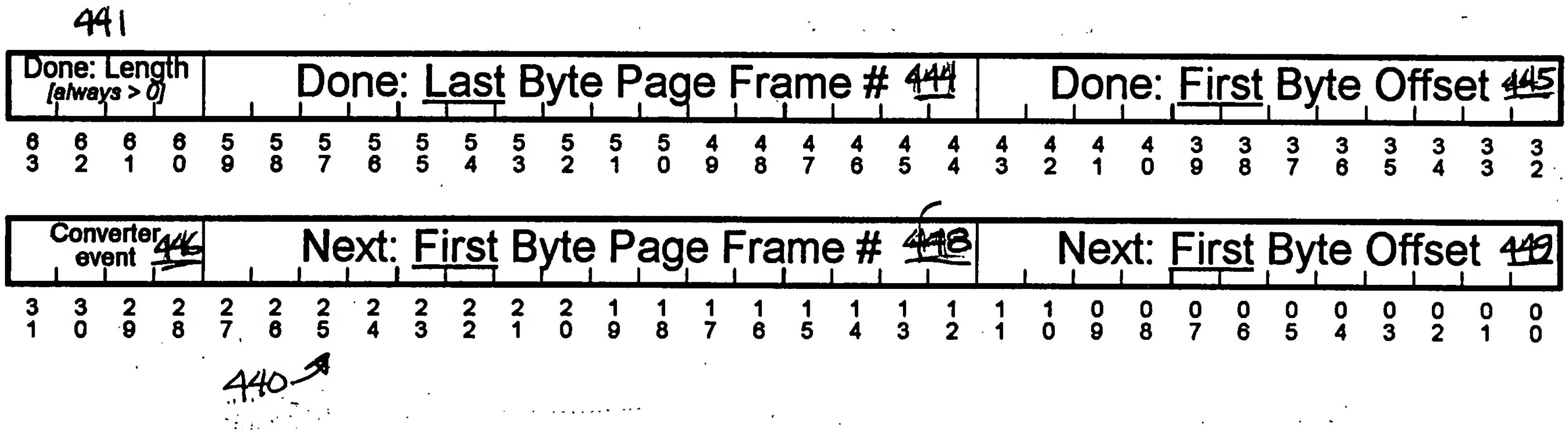


Fig. 4d *Near_Edge profile trace-packet entry*

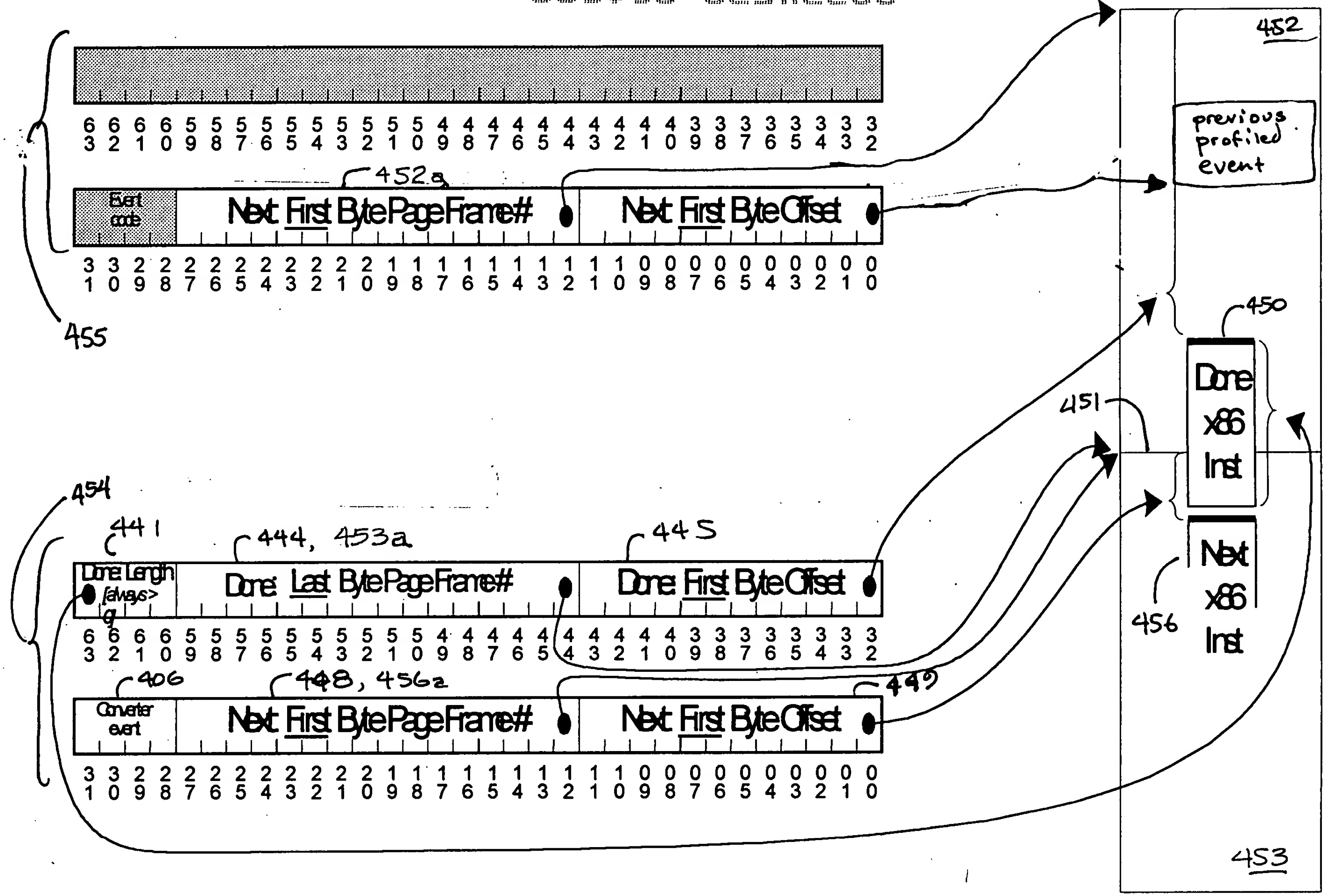

452
previous profiled event
455
Event code
Next First Byte PageFrame#
Next First Byte Offset
452a
6 6 6 6 5 5 5 5 5 5 5 5 5 5 4 4 4 4 4 4 4 4 4 4 3 3 3 3 3 3 3 3
3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2
3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
450
Done x86 Inst
451
Next x86 Inst
456
454
441
Done Length (always> 9)
444, 453a
Done: Last Byte PageFrame#
445
Done: First Byte Offset
406
Converter event
448, 456a
Next First Byte PageFrame#
449
Next First Byte Offset
453

Fig 4e

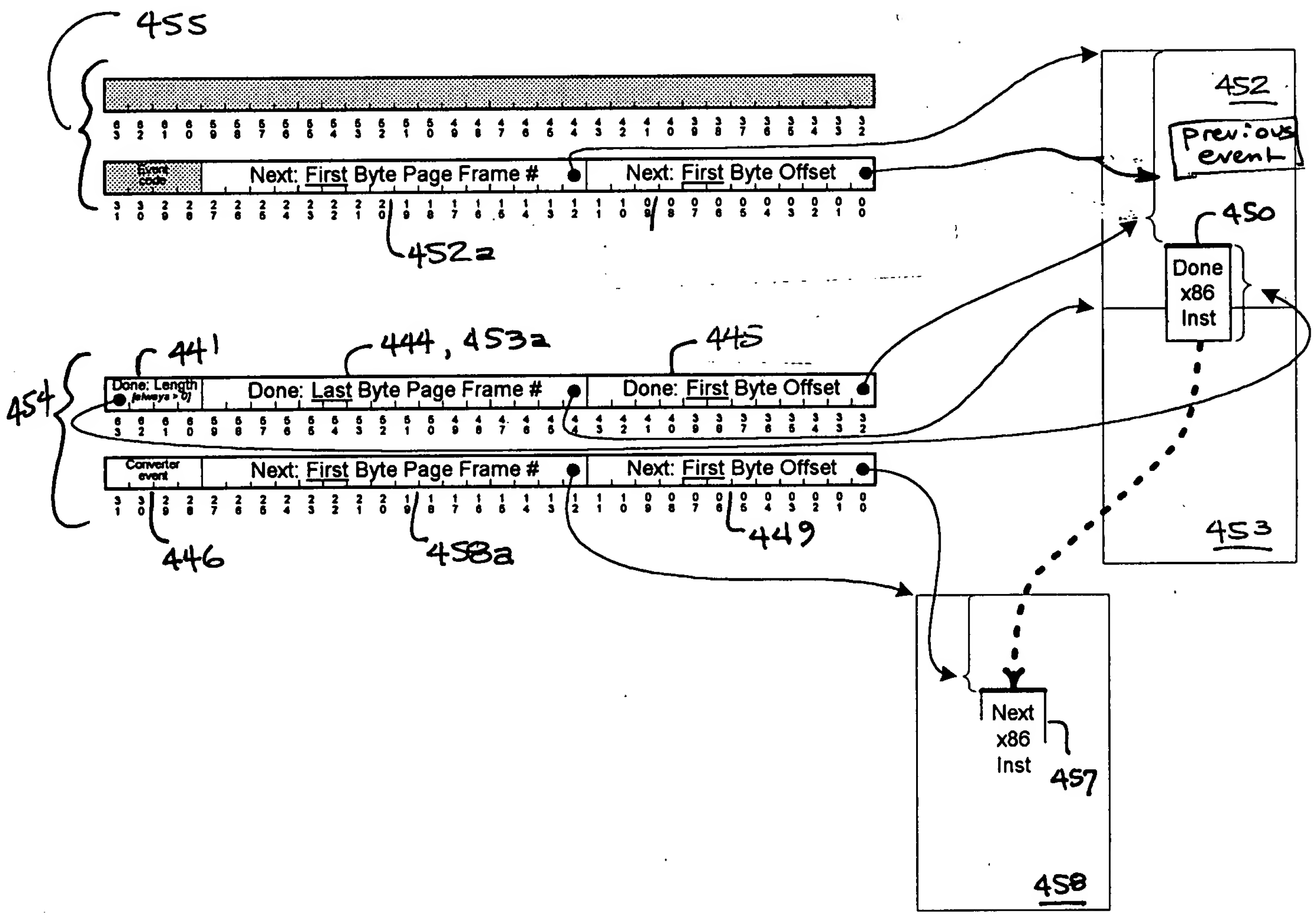

455
Next: First Byte Page Frame #
Next: First Byte Offset
Event code
452a
441
444, 453a
445
454
Done: Length (always > 0)
Done: Last Byte Page Frame #
Done: First Byte Offset
Converter event
Next: First Byte Page Frame #
Next: First Byte Offset
446
458a
449
452
Previous event
450
Done x86 Inst
453
Next x86 Inst
457
458

Fig. 4f

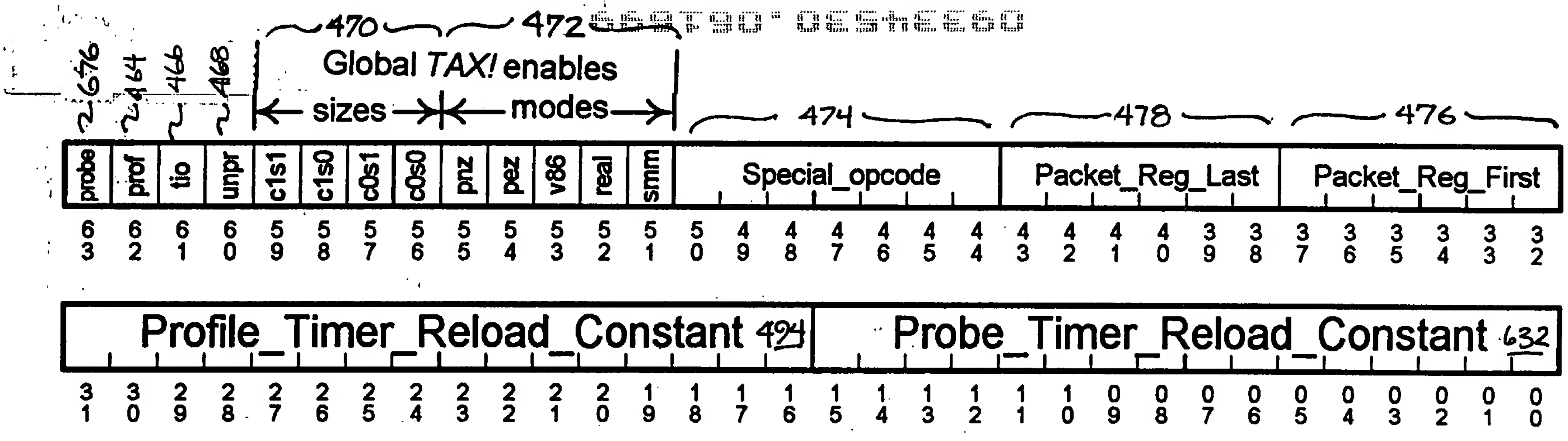


Fig. 4g Taxi_Control processor register 460

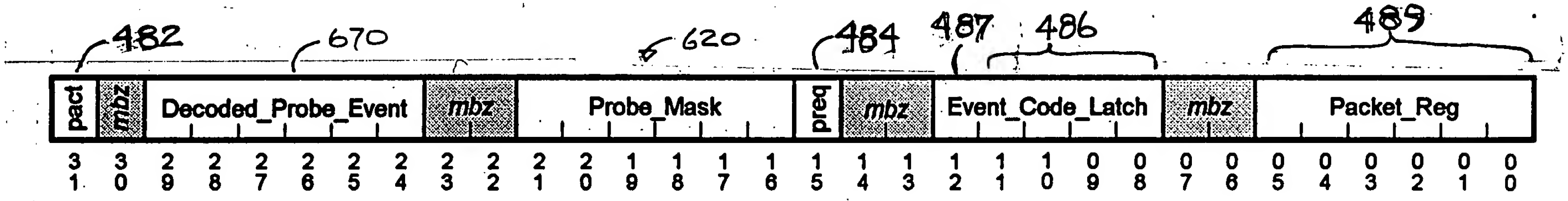


Fig. 4h Taxi_State processor register 480

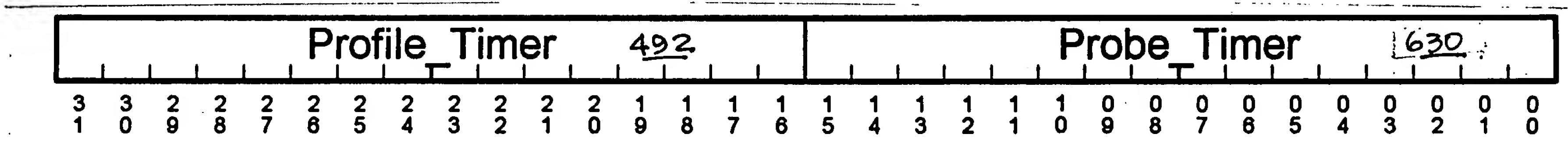


Fig. 4i Taxi_Timers processor register 490

Events
pe_{init} = "initiate packet" profile event 418
$pe_{\overline{init}}$ = non-"initiate packet" profile event
pe = any profile event 416
te = timer expiry
ap = abort packet

State Variables
PR = *Profile_Request* flag 484
PA = *Profile_Active* flag 482

Rules
te event ⇒
PR ← 1

PR & $\overline{PA}$ & pe_{init} ⇒
PR ← 0
PA ← 1
Init Packet_Reg
Save timestamp
Log event (CAP)
Full packet? / Packet_Reg++

PA & pe ⇒
Log event (CAP or NE)
Full packet? / Packet_Reg++

full packet ⇒
PA ← 0
Profile exception

ap event ⇒
PA ← 0

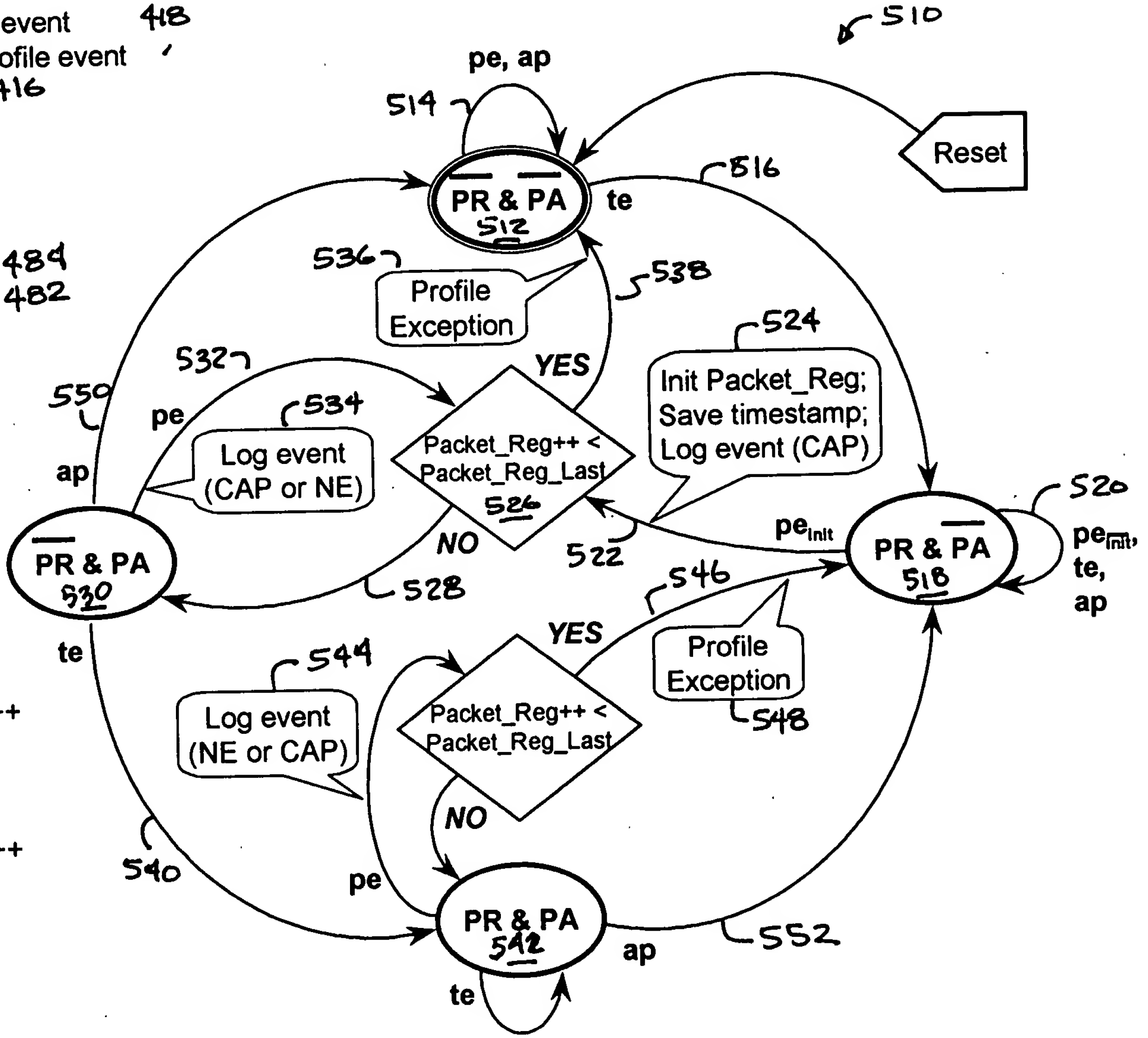
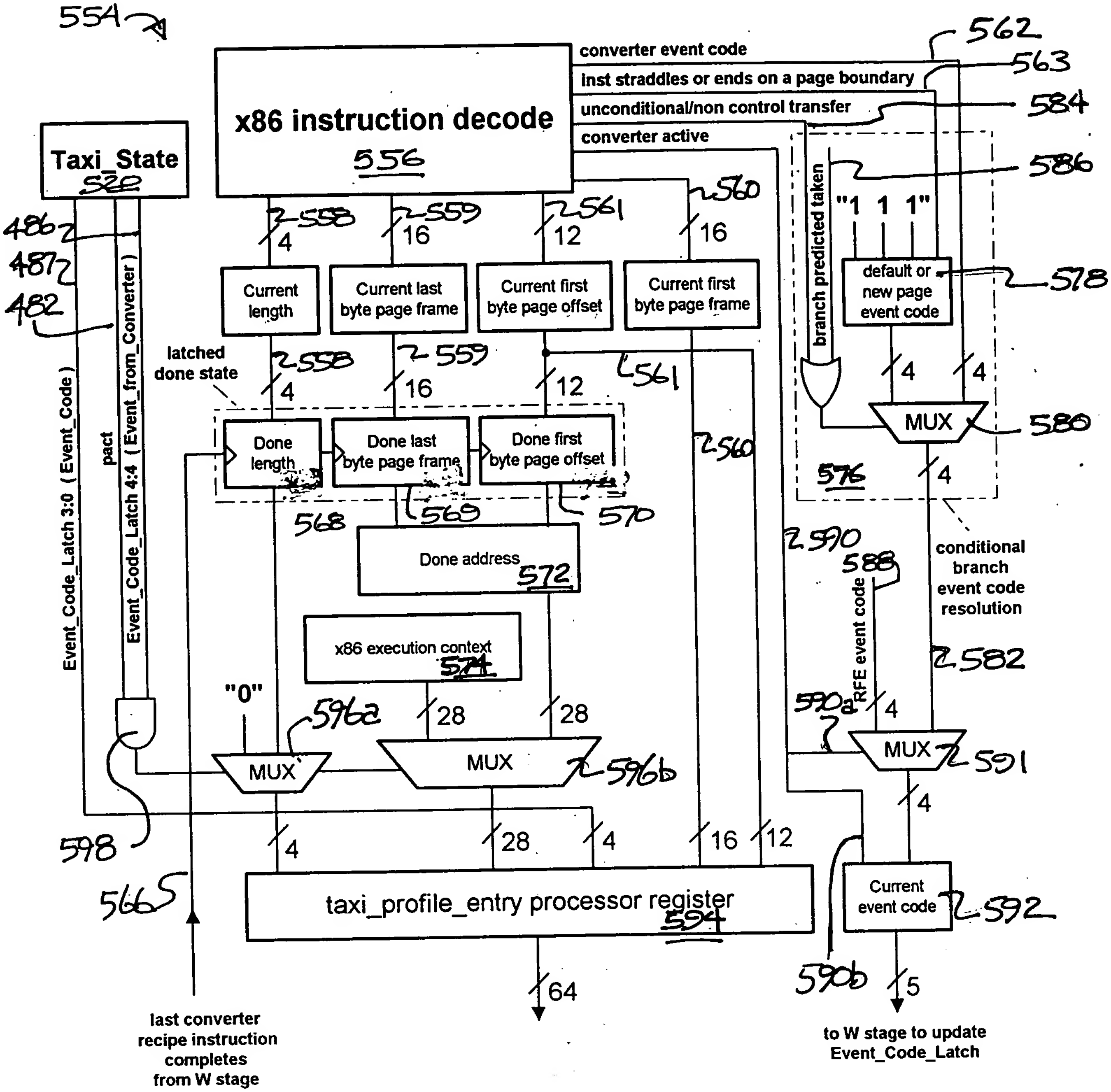


Fig. 5a

taxi profile entry generation



554
x86 instruction decode
556
converter event code
inst straddles or ends on a page boundary
unconditional/non control transfer
converter active
562
563
584
586
Taxi_State
520
486
487
482
Event_Code_Latch 3:0 (Event_Code)
pact
Event_Code_Latch 4:4 (Event_from_Converter)
558
4
559
16
561
12
560
16
Current length
Current last byte page frame
Current first byte page offset
Current first byte page frame
latched done state
558
4
559
16
12
561
Done length
Done last byte page frame
Done first byte page offset
568
569
570
560
branch predicted taken
"1 1 1"
default or new page event code
578
4
4
MUX
580
4
576
conditional branch event code resolution
Done address
572
x86 execution context
574
590
588
RFE event code
582
590a
4
"0"
596a
28
28
MUX
MUX
596b
MUX
591
598
4
28
4
16
12
4
565
taxi_profile_entry processor register
594
Current event code
592
590b
64
5
last converter recipe instruction completes from W stage
to W stage to update Event_Code_Latch

Fig. 5b

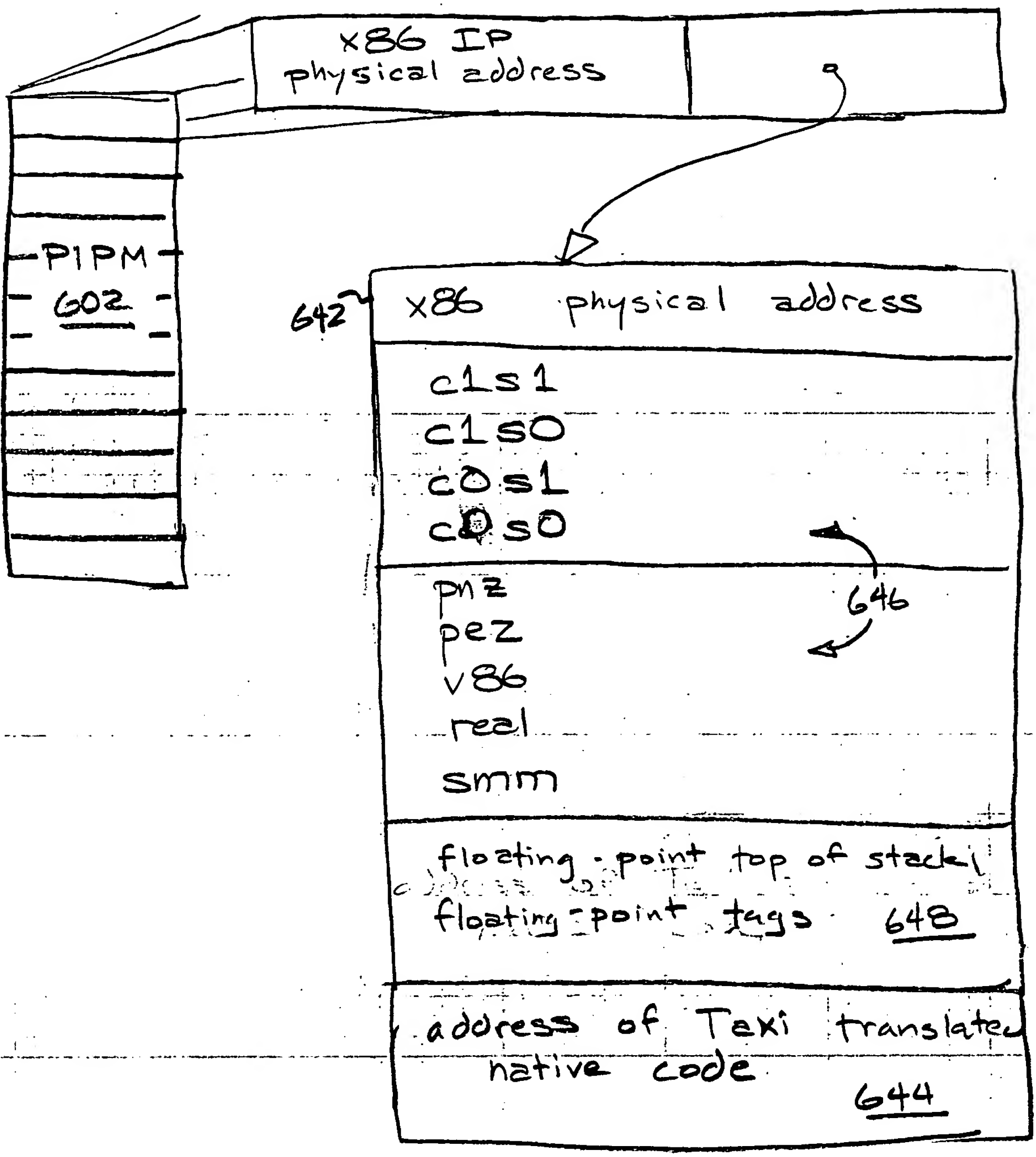

x86 IP
physical address
PIPM
602
642
x86 physical address
c1s1
c1s0
c0s1
c0s0
646
pnz
pez
v86
real
smm
floating-point top of stack
floating-point tags
648
address of Taxi translated
native code
644

Fig. 6a

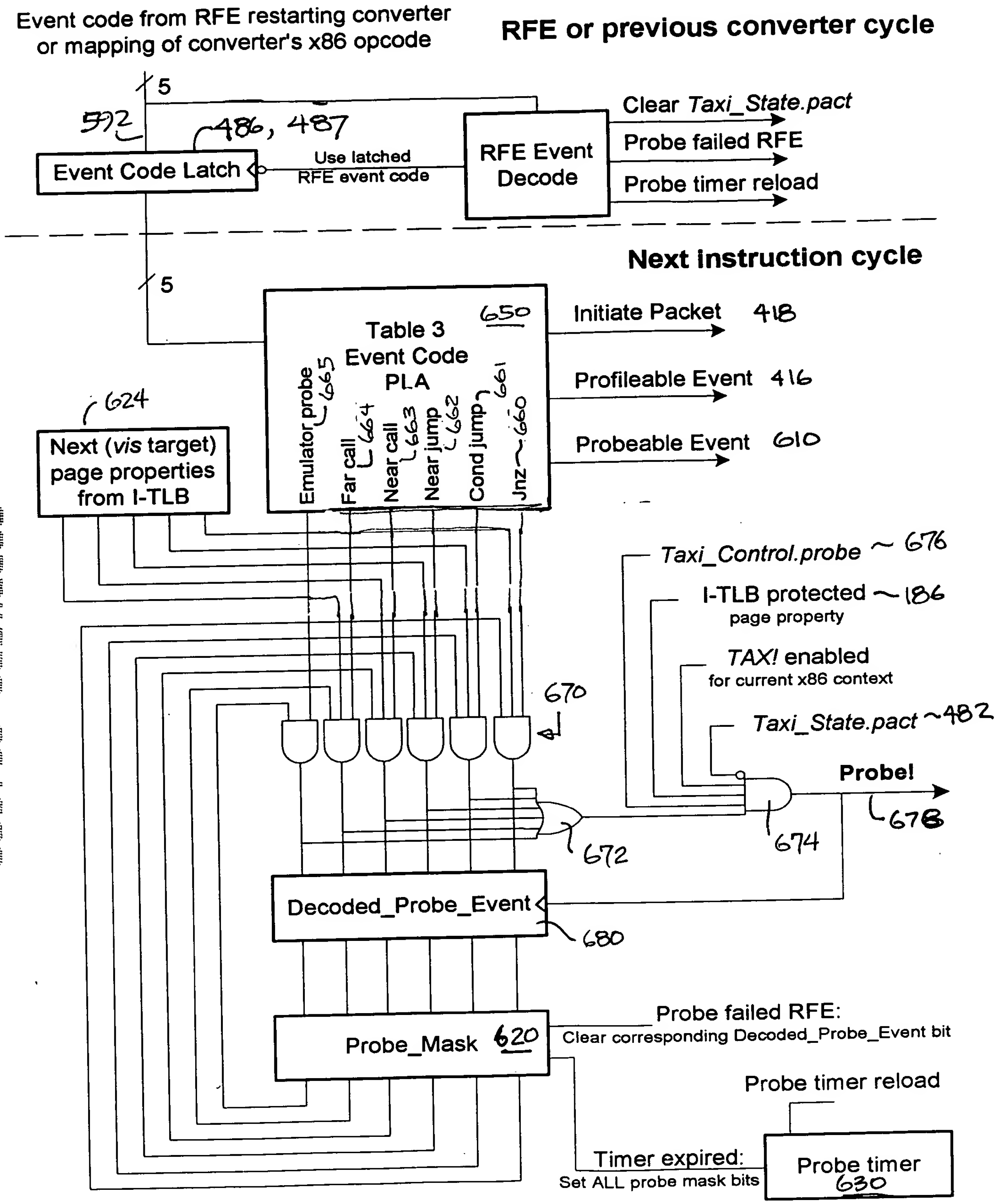

Event code from RFE restarting converter or mapping of converter's x86 opcode
RFE or previous converter cycle
5
592
486, 487
Event Code Latch
Use latched RFE event code
RFE Event Decode
Clear *Taxi_State.pact*
Probe failed RFE
Probe timer reload
Next instruction cycle
5
Table 3 Event Code PLA
650
Emulator_probe
665
Far call
664
Near call
663
Near jump
662
Cond jump
661
Jnz
660
Initiate Packet
418
Profileable Event
416
Probeable Event
610
624
Next (*vis* target) page properties from I-TLB
Taxi_Control.probe
676
I-TLB protected
186
page property
TAX! enabled
for current x86 context
Taxi_State.pact
482
670
Probe!
678
672
674
Decoded_Probe_Event
680
Probe_Mask
620
Probe failed RFE:
Clear corresponding Decoded_Probe_Event bit
Probe timer reload
Timer expired:
Set ALL probe mask bits
Probe timer
630

Fig. 6b

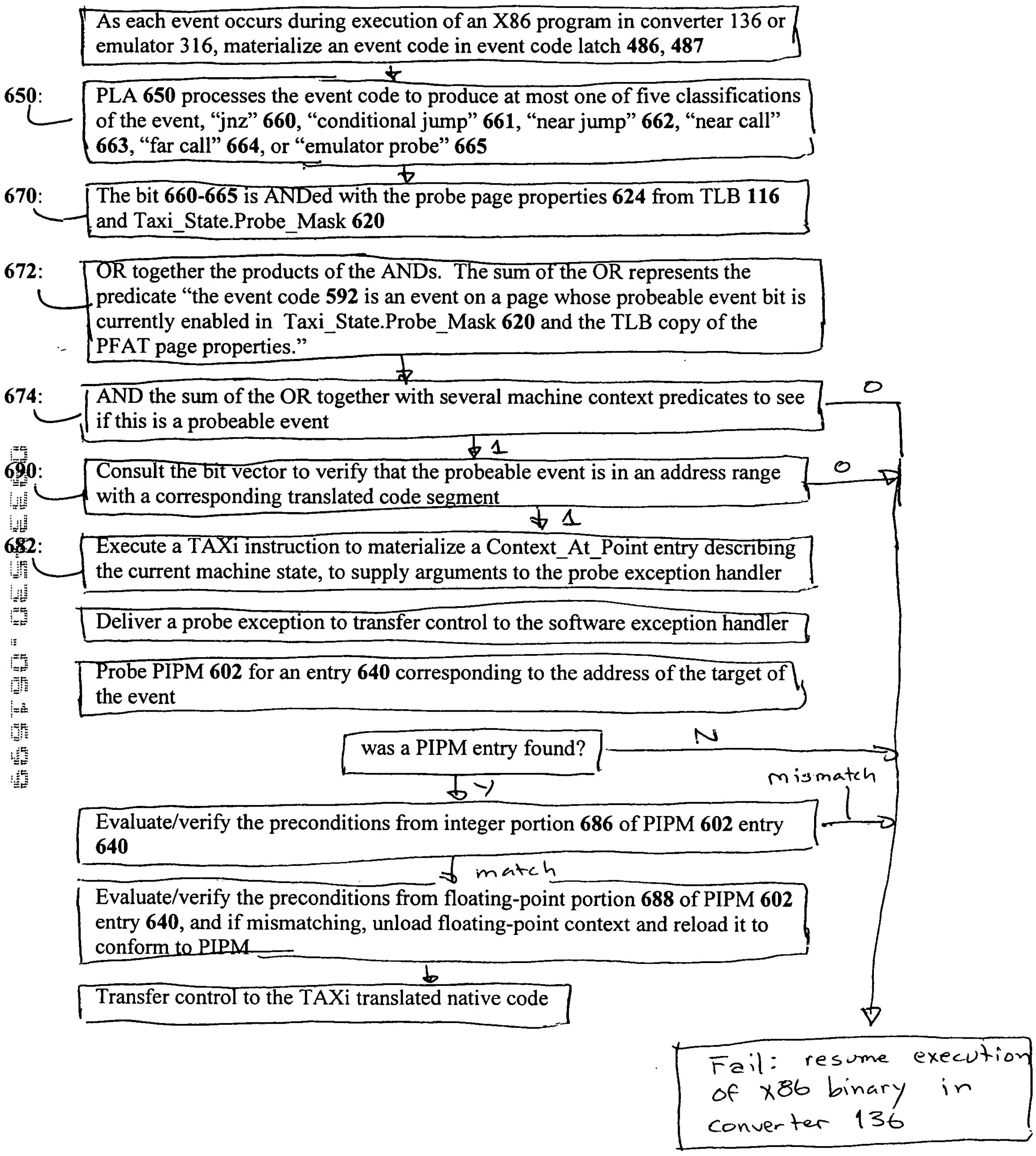

As each event occurs during execution of an X86 program in converter 136 or emulator 316, materialize an event code in event code latch 486, 487
650:
PLA 650 processes the event code to produce at most one of five classifications of the event, "jnz" 660, "conditional jump" 661, "near jump" 662, "near call" 663, "far call" 664, or "emulator probe" 665
670:
The bit 660-665 is ANDed with the probe page properties 624 from TLB 116 and Taxi_State.Probe_Mask 620
672:
OR together the products of the ANDs. The sum of the OR represents the predicate "the event code 592 is an event on a page whose probeable event bit is currently enabled in Taxi_State.Probe_Mask 620 and the TLB copy of the PFAT page properties."
674:
AND the sum of the OR together with several machine context predicates to see if this is a probeable event
0
1
690:
Consult the bit vector to verify that the probeable event is in an address range with a corresponding translated code segment
0
1
682:
Execute a TAXi instruction to materialize a Context_At_Point entry describing the current machine state, to supply arguments to the probe exception handler
Deliver a probe exception to transfer control to the software exception handler
Probe PIPM 602 for an entry 640 corresponding to the address of the target of the event
was a PIPM entry found?
N
Y
mismatch
Evaluate/verify the preconditions from integer portion 686 of PIPM 602 entry 640
match
Evaluate/verify the preconditions from floating-point portion 688 of PIPM 602 entry 640, and if mismatching, unload floating-point context and reload it to conform to PIPM
Transfer control to the TAXi translated native code
Fail: resume execution of X86 binary in converter 136

Fig. 6c